The Amazing Composobot:
Music Information Retrieval and Algorithmic Composition

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Marcus Walker

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

MARSHALL HAMPTON

May, 2018

Acknowledgements

Thanks beyond reasonable enumeration (pictured below) to Dr. Tracy Bibelnieks, Dr. Ted Pedersen, Kathryn Walkowski, and my preposterously ideal adviser, Dr. Marshall Hampton.

• • •

Figure 1: Thanks beyond reasonable enumeration

Dedication

For the voices whose improvised melodies elevate my progressions to accompaniment.

Abstract

Music has powerful and inscrutable effects on the human mind, and we are far from fully understanding how that magic works. But music is not random: there are patterns in the sounds and rhythms of a piece that can be analyzed, things that can be learned! In this work I will review relevant research on the subject of Music Information Retrieval and then introduce Composobot, an original program that incorporates and extends the lessons of that research. Together we will examine how Composobot prepares musical pieces for processing, analyzes them to extract systems of patterns and dependencies, and then composes novel musical pieces based on what it has learned. Finally, we will discuss how much of the magic that is in the music we love can be captured by learning patterns the way Composobot does, and how those methods might be tweaked to capture an even greater share of it.

Contents

List of Figures

Chapter 1

Introduction

Music is great. Rather, great music is great. Great music is fantastic- it's like stepping into a soul larger than your own, or stepping into the larger soul you didn't know was your own already. Really great stuff. Awful music, on the other hand, is awful. When it comes to awful music, we'd count ourselves lucky to only be listening to random, purposeless noise. We, humans, know how to tune out random noise. The alternative, noise arranged in patterns that actively capture our attention and offend us, is a thousand times worse. I hate it. Yuck!

Isn't that great in itself, though? What power music has to affect our seemingly inviolate inner states! Even when the effect is to make us cringe and grind our teeth, we might stop to marvel at the ability of a simple pattern of sounds to grab us so forcefully by the collar and abuse us so roughly. What strange locks must exist, unsuspected, in our minds, waiting only for the right pattern of sounds to turn their tumblers and open unanticipated vistas of feeling and experience!

Before we begin to explore that particular rabbit hole, let's address a question we should all be asking ourselves at this point in a discussion of the potency of arranged sound: doesn't this not matter at all? We have a thousand straightforward examples of patterns of sound affecting our inner states in ways that aren't surprising at all: human speech! If you say, "please give me that glass of water" or "you're quite pretty", those patterns of sound have predictable and straightforward effects on my inner state for the very simple reason that those patterns of sound are messages encoded in sound, and I know exactly how to decode the sounds and understand the messages. I could even

restate the same messages using totally different sounds: “surrender thine encapsulated hydration unto my possession”, or “your face is among the best faces I have ever seen”, not that I recommend communicating with humans in either of those ways.

Perhaps unsurprisingly, since this is a one-sided conversation I’m able to engineer any way I like without interruption, I’ve led us into a trap! It’s not too great a leap of analogy to imagine that music, too, encodes some kind of abstract message that we know how to decode the same way we decode language. There would be some leftover questions about the nature of the message and whether we’ve learned to decode it in some way analogous to the way we learn language (I would argue that we, as individuals, do not learn to understand music that way), but it is not an unimaginable analogy. However, as we have seen, we know how to take the messages of speech and re-encode them in entirely novel, if awkward, patterns of speech-sounds. With either pattern of sounds given as examples above, a careful listener with a thesaurus on hand should deduce that the speaker wants me to hand them a glass of water, or that they like the way I look.

It could maybe go without saying that this has nothing to do with performing some kind of phonemic pattern recognition on a spoken sentence and detecting a certain ratio of consonant to vowel sounds, or a preponderance of “ee” sounds within a phrase, and deducing meaning from that. We don’t care about any of that. We are barely considering the raw sound level of the messages at all when we decode them, except insofar as those sounds correspond to a dictionary of meanings in our minds. When we think about the messages of speech in a language we understand, we think about words and meanings, not the sounds themselves.

The flaw in our analogy, then, is laid bare: the messages of music are considered at the raw sound level, because there is no dictionary of meanings that the sounds of music correspond to (unless you are a spy using a piano to transmit coded messages). To illustrate, imagine your favorite (or least favorite!) melody. Now, try to restate that melody using entirely different notes and rhythms. In some sense, you might argue, a talented composer could do just that, but I think that if you ask yourself whether a third party, hearing those two melodies and without prompting, would recognize that they convey exactly the same message, we will reach the same conclusion: a melody doesn’t have a literal meaning the way a sentence does. A direct analogy between the

way the sounds of speech affect our thoughts and the way the sounds of music do is insufficient to explain how music performs its grand magic on us.

Even that statement is misleading, though. We are not simply trying to explain how music performs its magic on us. We could say that the magic of music enchants and ensorcels us, but music does not ensorcel a rock or a tree stump the same way. Half of the magic is being performed by us, the listener! The potential music has to affect us is extraordinary, but our potential to be affected by music is at least as extraordinary!

It probably isn't possible to understand how music affects us without understanding ourselves- in fact, understanding ourselves is probably the most important reason to study music, outside of the sheer pleasure of it. To state it differently: a musical analysis is really a cognitive analysis of what parts of music a mind recognizes and responds to. A better understanding of the patterns in music that affect us, the patterns that most people recognize and react to without any training or intention, informs a better understanding of how our minds work.

This work before you has two purposes. The first is to explore what patterns in music are crucial to its magic and test that exploration by generating music from those patterns and seeing whether the magic is still there; the second is simple pleasure. Neither is more important than the other- in fact, much like the enchanting intertwining of music and the mind that hears it, neither would be doing magic without the other. And if work is going to have just one purpose, it ought to be magic.

1.1 Overview

The heart of this work is The Amazing Composobot, a charming program that extracts patterns and relationships from existing music and uses what it learns to compose new music! Briefly, Composobot reads symbolic music in the form of MIDI (Musical Instrument Digital Interface) files; analyzes all of the pieces they read to learn how to build chord progressions, accompaniment rhythm patterns, and melodies; composes entirely novel musical pieces based on the patterns they learn from the music they analyze; and outputs the resulting compositions as MIDI files that can be listened to and enjoyed by one and all.

The details of this process are myriad and interesting, and it is the privilege of

this document to describe them. We will begin in Chapter 2 by discussing some of the problems and relevant research on the subject of music information retrieval, the learning of patterns from music; we will also offer a brief primer on Markov processes and Hidden Markov Models in Chapter 3 that will aid in understanding the methods Composobot uses. Once that is all out of the way, we can get to the fun part: examining Composobot's methodology in detail! Chapter 4 will give an overview of their scope and major components. Chapter 5 focuses on Composobot's implementation in code and their preprocessing of the music they analyze. Chapter 6 will detail the learning process, and Chapter 7 will explore the arcane mysteries of the algorithmic composition step.

Composobot's unabridged source code, a description of the training corpus, and example output of pieces Composobot has composed can all be found in the appendices of this document. Thank you for reading and, above all, enjoy!

Chapter 2

Music Information Retrieval

I hope that we are can agree that music isn't totally random. If we play a quarter note of middle C on the piano, it would be silly to expect, with equal probability, any possible note at any possible duration to come next. If the next note is a quarter note of the E above middle C, we won't be too surprised. If the next note is $C\#$ at the highest register audible to humans, played for a duration of 6 hours, that's a little unexpected! There is some kind of pattern of interactions between events in music, some web of dependencies. There is information of some kind there.

Music information retrieval, then, is the hunt for it: the process or enterprise of finding the information present in music and retrieving it for some kind of analysis.

In this chapter we'll take a brief look at the parts of the history and current research on music information retrieval that are relevant to understanding how Composobot works and why it does things the way it does. My hope is that you, my dear reader, walk away from this chapter with the conceptual foundation necessary to fully understand what Composobot is doing when its inner workings are discussed in detail in later chapters.

2.1 Representation

There are a wide variety of ways musical information can be represented, and we could all probably come up with a handful of ideas off the top of our heads. Sheet music likely comes to mind. A guitarist may think of tablature, a representation using the strings of

a stringed instrument rather than the staff of traditional sheet. Non-musicians may be concerned at this point that they misunderstood the question, because they immediately thought of records or mp3's instead of pieces of paper- you haven't! A home video of a children's choir, a roll from a player piano, even the waveform etchings on the Voyager spacecrafts count. If the thing in question can be decoded such that a musical pattern of sounds makes it to your brain, then it's a representation of musical information.

Rather than try to explore each possible representation one-by-one, I'm going to focus on the broadest category of representations that is relevant to this work: discrete symbolic representations (e.g., sheet music). In particular, we will look at digital discrete symbolic representations that encode individual notes with pretty minimal information: the pitch (e.g., $A5$ or $C\#3$), the onset time (i.e., the point in time in a piece where a note starts playing), and the duration (i.e., the length of time between a note's onset and "offset", when it stops). Even more particularly, we're going to look at the MIDI (Musical Instrument Digital Interface) representation, which is what Composobot reads and, therefore, is the only thing we care about at all.

MIDI was devised as, and technically is, a communications protocol for transmitting music information and control signals electronically. It is a standard format, and its use ensures that electronic devices can communicate music information to one another effectively and without data loss. As such, it consists of more than just the note-level information described in the previous paragraph- MIDI format also encodes metadata about a piece's speed and time signature and format; divisions of notes into any number of tracks and up to sixteen audio channels; and even event data to control changes in tempo, volume, instrument, vibrato, etcetera[1]. While this is certainly interesting and useful in a lot of contexts, we don't care about most of that stuff and are going to abstract it away. Exceptions will be that every piece in our corpus will be encoded in MIDI such that melody and accompaniment voices are stored in separate tracks so that they are easy to separate, and will have a uniform tempo defined by a "time division" of 96 milliseconds. That is to say that a quarter note, for any of our corpus pieces, will have a duration of exactly 96 milliseconds.

More important for our purposes is the note level information MIDI offers, because it includes exactly what we want: pitch, position, and duration. Each is represented by an integer value. Pitch is defined such that $C5$ (middle C) $= 60$, and a note n half-steps

up or down from $C5$ is equal to $60 + n$ or $60 - n$, respectively. For example, $B3$ is thirteen half-steps below middle C, and so has the value of $60 - 13 = 47$. Position and duration are defined in terms of milliseconds: number of milliseconds into a piece a note occurs, and number of milliseconds a note lasts, respectively.

Let us imagine a note that has the pitch $C5$ (middle C), occurs at the beginning of the second measure of the piece, and is one half-note in length. Given that we've specified the duration of a quarter note as 96 milliseconds and assuming our piece is in 4/4 time, the note would be represented as: $\{60, 384, 192\}$. $C5 = 60$ as described in the previous paragraph; $384 = (96)(4)$, which represents that the note occurs four quarter notes into the piece, which is where the second measure begins; $192 = (96)(2)$, the duration of two quarter notes, which is equal to the duration of a half-note.

For the purpose of this work, imagine MIDI files as consisting of two sequential lists of notes defined as above: one list for the melody notes, and another for all of the other notes. Sometimes those lists will be considered in combination, as one giant bag of notes. At other times, they will be considered separately. The important thing to keep in mind is that they are ordered lists of note pitches, positions, and durations.

2.2 Key-Finding

If you're going to learn general musical patterns from a piece of music, it's really helpful to know what key it's in. Contrary to the unlimited analogy principle of colloquial goose wisdom, what's good for C major is not good for $F\#$ minor. If we generate music using the lessons of every key all mixed together indiscriminately, then a little consideration reveals that we might generate any note at any time, and random noise is the most merciful thing we might inflict on our poor listeners who, let us assume, have done nothing wrong.

So when our program analyzes a piece to learn its patterns, we want it to know the key of the piece it is analyzing. It should be able to say "these are C major patterns, I'm going to keep these separate from the $F\#$ minor patterns". It would be possible to hand-annotate each input piece beforehand with its key and mode ("C" and "major" or "$F\#$" and "minor"), but a few things have to be true in order to be able to do so accurately. First, any given piece in our corpus, or set of training pieces, must have

an agreed-upon key, declared explicitly by a composer or music theorist and accepted broadly to whatever extent our definition of accuracy demands. That's usually true, but there are counterexamples- what if we want to analyze something that itself was algorithmically composed? Second, and most crucially, we have to know what corpus we're going to use and have the time and resources to perform the hand-annotation! Every time we want to analyze some new piece, we'll need to stop and determine its key and make sure to tell our program what to expect. Wouldn't it be nice if there were some relatively straightforward and reliable way to annotate a piece algorithmically?

Fortunately, there are reliable ways to determine a piece's key algorithmically, depending on the strictness of your definition of reliable! Work on this goes back a long way; in some respects, it is as old as music theory itself. The early computational models, such as the work of Longuet-Higgins & Steedman [2] in 1971, relied on the most fundamental music theoretical insights into the nature of keys: which set of notes define the key. A piece could be scanned note-by-note from beginning to end such that, initially, all keys are possibilities. With each scanned note, any key not containing that note would be discarded as a possibility. At the end of the piece, if there is a single key possibility remaining, the model selects that key. If there is more than one remaining, it selects the key whose tonic is the first note of the piece. If none remain, as often happens with more complex pieces containing accidentals, it again selects the key whose tonic is the first note of the piece in spite of that key having been eliminated along the way.

As in most areas of modeling, work has since shifted away from a strict rule-based approach toward probabilistic or mixed rule-probabilistic modeling. The Krumhansl-Schmuckler key-finding algorithm[3] introduced in 1990 relies heavily on the concept of "key profiles", introduced by Krumhansl & Kessler in 1982[4]. A key profile represents, in some way or another, the expected distribution of notes in a piece of a given key. Early key profiles built by Krumhansl & Kessler were developed by playing a short key-establishing piece for an audience of human subjects and then playing a single additional note. Listeners were asked to rate, on a scale of 1 to 7, how well they thought that additional note "fit" with the piece they had just heard. For each key, each of the twelve notes was then weighted according to the scores it had received from the human subjects, and the resulting distribution became the key profile for that key. The profiles generated by Krumhansl & Kessler were found to very closely match insights of music

theory: the largest probability space is given to the tonic, followed by tonic triad notes and then the other notes that define that key, with chromatic notes receiving the least probability space.

An interesting and useful observation arose from the development of these key profiles, which also matched the insights of music theory: within a mode (that is, major or minor), the distributions of different keys looked very similar. The key profile for *D* Major, for instance, looks a lot like the key profile for C Major, only shifted up two notes. Incorporating this assumption of intra-mode transpositional similarity not only simplifies the process of building key profiles by allowing the key profiles for each key in a mode to be generated by building a single key profile in that mode, it allows for more robust models requiring fewer data, since pieces in any major or minor key can be analyzed and their results transposed to the same key. In other words, an analysis of five pieces in *C* Major, five pieces in *D* Major, and five pieces in *F#* Major is really an analysis of fifteen pieces in "Major". A greater volume of representative observations helps build a more robust model.

Subsequent work by, for example, Temperley[5] and Hu & Saul[6] embraces increasing computational power and availability of data to build key profiles based entirely on the observation of pieces, without the input of human subjects. Building such profiles requires some corpus of key-annotated pieces, but the process is relatively simple: observe the frequency of notes occurring in a piece of a given key, and calculate the probability distribution directly from those frequencies. The key profiles generated by Hu & Saul[6] are similar to the profiles generated by Krumhansl & Kessler[4], but model starker differences in probability between in-key and chromatic notes.

Figure 2.1: Krumhansl-Kessler key profiles.

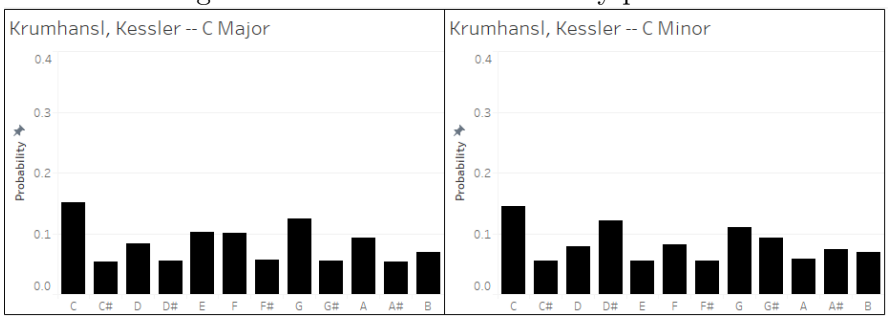
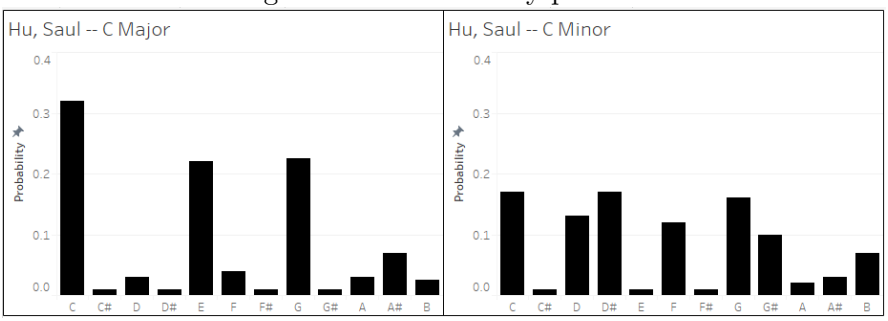


Figure 2.2: Hu-Saul key profiles.



Using a set of key profiles to classify a piece into a key, then, is at its heart a process of comparing the observed note distribution of a piece against the note distributions of the key profiles and selecting whichever key's profile is most similar to the distribution of the piece. The method of key-finding used by Composobot relies heavily on this process and on the Hu-Saul key profiles, and is described in detail in Section 6.2.

2.3 Chord-Labeling

In monophonic music, we might get away with modeling a musical piece by simply considering what notes are likely to be played in succession- that is, the probability of some note n being played given some note m being played directly beforehand, or given some set of notes $\{m_1, \cdots, m_n\}$ being played in succession beforehand. If music were always a series of non-overlapping notes played in succession, then this might be a reasonable level of information to target for retrieval.

Of course, music isn't always monophonic. A saxophone solo is; a piano concerto likely is not. For polyphonic music we must consider not only the succession of notes, but also the concurrence of notes. Why are some notes played together, or in close metric proximity to one another, while others are not? What determines the set of notes that are likely to be played during a given chunk of metric time, and what determines which set of notes are likely to be played during the next such chunk?

It is for these reasons that we consider chords and the progression of chords. Chords, as I've implied, are not strict sets of entirely overlapping notes played at once, but rather represent some distribution of likelihood of notes being played during a span of time.

Paiement, Eck, & Bengio[7] articulate this idea eloquently:

> In general, the chord progression itself is not played directly in a given music composition. Instead, notes comprising the current chord act as central polarities for the choice of notes at a given moment in a musical piece. Given that a particular temporal region in a musical piece is associated with a certain chord, notes comprising that chord or sharing some harmonics with notes of that chord are more likely to be present.

The aim of chord-labeling, then, is to divide a piece into "temporal regions" and then find some way to characterize the "central polarities" of those regions. A labeled chord progression should describe a progression in divisions of metric time through a piece, describing in some fashion how likely given notes are to appear in those divisions.

2.3.1 Chord Characterization

The tough and central question is of how we want to characterize a chord. The number of ways to do so is infinite; even among those finitely many characterizations that are of at least some δ usefulness, the possibilities are remarkably varied. Rather than explore all of them, let's discuss two approaches that more-or-less describe the range of the space of useful characterizations.

The first and, probably, most familiar of these characterizations is a symbolic aggregate description: a short series of symbols that describe a set of possible notes in terms of a scale. Chords described in this way usually consist of the letter of the chord's tonic (e.g., "A" or "$F\#$"), a symbol describing the modality of the chord ("m" for minor, "M" for major, "aug" for augmented), and any interval modifications or additions to the basic 3-note triad described by the preceding symbols ("7" for including the seventh of the scale, "$/A$" for including an A below the tonic as a bass note). Examples include "Am" for A minor, "$CM7$" for C major seventh, or "$F\#aug$" for an augmented F# triad.

This characterization has a few advantages and a few disadvantages. It has the advantages of being compact, requiring just a few symbols to communicate a lot of information about what notes are likely in a given time division, and of having a relatively small range of possible values- small enough that it is not unreasonable that you

could enumerate all likely chords in a list. However, its compactness is the result of it relying heavily on the music theoretic acumen of its interpreter. It certainly wouldn't be straightforward for a non-musician to translate "$DmM9$" into accurate knowledge about the distribution of notes in a time division so labeled. In particular, there is a probability distribution of possible notes implied symbolically that a musician may understand implicitly, but there is no precise definition of that distribution accompanying the symbols.

This representation's small range of values, similarly, implies a disadvantage: it does not capture information about particular voicings of a chord. Octave is abstracted, meaning that the note sets $\{A0, E1, C1\}$ and $\{A5, C6, E7\}$ will both map to "Am". Many different note sets are aggregated to a small set of symbols, which is great for storage of information and reduction of dimensionality, but makes it impossible to reconstruct the details of a chord voicing accurately.

On the other side of this range is the representation adopted by Paiement, Eck, & Bengio[7], authors of the previously quoted definition of a chord, in 2005. Their chord-labeling model characterized chords entirely literally: a chord would be defined by the exact pitch and octave values of the notes within a time division. The note sets from the previous paragraph's example, $\{A0, E1, C1\}$ and $\{A5, C6, E7\}$, would be represented by the strings "$a0e1c1$" and "$a5c6e7$", rather than both mapping to "Am".

This representation is not without its own disadvantages. The first is that there is no way to reasonably enumerate the possible chords beforehand. Even limiting ourselves to the 88 likely note values in piano music, enumerating all possible combinations of any number of values is combinatorially apocalyptic. This is the inverse of one of the challenges faced by the symbolic aggregate representation: it is always possible to perfectly reconstruct a chord voicing, but the storage requirements for a chord progression are little less than that of a whole piece.

A more informatically relevant disadvantage is that learning becomes sparse. Probably there are quite a few qualities in common between "$a0e1c1$" and "$a5c6e7$", both being voicings of Am, but they are learned as entirely separate entities- any underlying similarities are missed. Given corpora of the same size, we might learn quite a bit less about the general class of A minor chords using a literal representation, and may risk overfitting the particular measures and pieces we observed these chords in, if there are

relatively few examples.

There is no perfect answer. The characterization appropriate to an analysis depends on the motivations for that analysis. A study of common chord progressions will likely prefer a symbolic aggregate representation; a study of chord voicings will prefer a literal representation. The focus of this work is somewhere in between, and will correspondingly prefer a representation that falls somewhere between the two extremes.

2.3.2 Assignment of Labels

The point of discussing chord representations was that we wish to label time divisions with chords in some way. So, once some chord representation has been selected, a piece will have to be sliced up into time divisions and analyzed so that it can be labeled with a chord using the chosen representation's lexicon.

A naive, but generally effective, approach is to simply divide a piece into even slices (e.g., half-measure lengths) and label each based on the notes appearing in that slice. The division of the piece shouldn't be entirely arbitrary- if the slices are too thin, they will have too little evidence to meaningfully determine a chord. What chord does a single note of $A5$ represent? A Major? A Minor?

On the other hand, if the slices are too thick, then it's more likely we'll be trying to assign one chord to a section of music containing multiple chords in progression. Not only would this make it difficult to accurately assign any correct chord, since two chords combined may mimic some third chord, but we would certainly be losing information about the chord progression as a whole. One solution, again naive but generally effective, is to examine the pieces in a corpus and determine a reasonable division for that corpus.

The method for then assigning labels to those divisions depends strongly on the chosen representation. In most cases the methods are alike in that they involve calculating some measure of distance between the observed set of notes and each of the chords in the set of representations, then accepting the nearest such chord.

The simplest representation for determining a label is probably the literal representation of Paiement, Eck, & Bengio[7], described above. If a chord is defined as the set of all notes played in a time division, then labeling is trivial: simply label each time division with the notes that fall within it. This is essentially a calculation of distance in the discrete metric: identity implies minimum distance, and non-identity implies maximum

distance.

In the symbolic aggregate representation (e.g., $Am7$) it is possible to enumerate the space of possible chords in a list prior to analysis. For each enumerated chord, a set of pitch classes could be selected that define that chord, where "pitch class" is a note's pitch value with octave abstracted away- for example, the pitch values $A4$ and $A6$ both have pitch class A. There are twelve pitch classes in western tonal music, and so a chord might also be represented by some list of elements from those twelve classes: $Am7$ is an Am triad (A, C, and E) with an added minor 7th (G), and so might be represented by $\{A, C, E, G\}$. Another way to represent the same list is with a binary 12-vector, where position 1 corresponds to the pitch class C and position 12 corresponds to the pitch class B. Then $\{A, C, E, G\} \equiv \{1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0\}$.

Then, for a time division, a similar 12-vector could be constructed simply by looking at all pitch classes present in the time division: a time division containing the notes $a5$, $c6$, $g6$, and $c7$ has pitch classes A, C, and G, and a vector representation of $\{A, C, G\} \equiv \{1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0\}$.

Distance could be calculated between a time division and each chord in any number of ways. The most straightforward would probably be the Hamming distance between the binary strings of the 12-vectors, or the number of positions at which those strings differ. In the case of the 12-vectors offered above as examples, that distance is 1.

Notice, however, that in our example time division the pitch class C appears twice, but is not given any additional weight in the 12-vector. That frequency information is abstracted away, and it's entirely possible that that information is relevant. Note also that the lowest note has a pitch class of A, and that that information is similarly lost. These abstractions may serve as motivation for applying some kind of weighting to the elements of the 12-vectors, rather than representing mere presence or absence with binary values. Indeed, we will explore a more nuanced weighting scheme in Section 6.1.

With a 12-vector of non-binary values, Hamming distance will no longer work. A straightforward generalization to 12-dimensional Euclidean distance can be, and often is, employed instead. This measure will prove essential to the way Composobot calculates distances between chords and determines chord labels and will also be discussed in detail in Section 6.1.

The final application of this measure of distance is, again, to find that chord in an

enumerated list of possible chords that is nearest to the notes that appear in a chosen time division. Once that chord is determined, the time division in question can be labeled with the chord's representation (e.g., $Am7$).

Regardless of representation, once each time division in a piece is labeled, a chord progression for that piece can be defined as an ordered list of chord labels: examples of short chord progressions include $\{a0e1c1, c3e3g3, a5c6e7g7, b5e6b7\}$ in the literal representation or $\{Am, CM, Am7, B4\}$ in the symbolic aggregate representation.

2.4 Melodic Phrase Boundary Detection

When examining the musical piece as whole, we considered the distribution of notes within a time division to make a progression of chords, rather than considering individual notes in sequence. Similarly, an examination of melody in terms of individual notes in sequence is too naive to extract most of the useful patterns that make a melody what it is. A melody is not just a string of musical notes in order; there is underlying structure that ties those notes together, and to the rest of the piece.

The appropriate level of abstraction at which to try to capture that structure is very much an open problem. That said, Temperley offers evidence to suggest that there is a level analogous to an existing music theoretical structure that captures a great deal of the information we want: the melodic phrase[8].

The definition of a melodic phrase, much like the definition of a chord, is not a rigorous one. A melodic phrase might be described as a sequence of notes in a melody that "belong together" in the sense that words in a sentence "belong together". That is to say that a listener might perceive some notes as beginning a sort of musical statement, and some notes a short while later as concluding it. This is an admittedly vague notion, but human listeners have a surprisingly easy time dividing melodies into phrases. It stands to reason that there are patterns we could exploit to make those divisions algorithmically.

Much of the work that has been done on the problem of detecting melodic phrase boundaries relies on a few intuitions about the way that human listeners draw these divisions. One such observation is that longer gaps between notes tend to denote divisions between phrases, which is to say that listeners tend to perceive melody notes

as being "together" when the gap between them is shorter, and "not together" when the gap between them is longer. Another is that the length of phrases is generally in the same range, which is to say that listeners are more likely to perceive a boundary between phrases after roughly a certain number of notes, which varies by genre. A third is that melodic phrases tend to start at roughly the same point in the metrical structure as one another, which is to say that listeners expect phrases to begin near strong and even beats of measures, and especially to begin near the same beats as the other phrases they've perceived so far.

Temperley synthesizes these intuitions into what he calls Phrase Structure Preference Rules, which are rules for assigning greater or smaller probability to a note that it marks the end of a melodic phrase, based on measures of the aforementioned intuitions. He calls those measures the Gap Rule, the Phrase Length Rule, and the Metrical Parallelism Rule. Exact implementations of these rules are not offered, but general formulae are suggested[9]. The general formulae and ideas behind Temperley's Phrase Structure Preference Rules are adapted and implemented within Composobot; the details of their use are described in detail in Section 6.5. For the time being, it's important to note that these rules are preference rules and not strict rules, which is to say that they increase or decrease the probability of a melodic phrase boundary appearing between notes, rather than dictating explicitly where boundaries fall.

Chapter 3

Markov Modeling

Here's the hard news: it's going to be tough to understand how Composobot transforms the information it retrieves from music into a model that can be used to generate new music unless you have a foundational understanding of Markov chains and Hidden Markov Models. But before you get too excited about having the perfect excuse to pack it in and head home, I've got a second bitter pill for you: I'm going to do my best to provide that foundational understanding right here in this very chapter. Unless you're willing to risk being slightly impolite, you've no recourse but to read on.

I'll begin with a brief explanation of the Markov assumption and how it is used to build the type of stochastic model called a Markov chain. Once we've got a firm grasp on those essentials, I'll discuss a particular type of Markov model used when the phenomenon being modeled cannot be observed but some of its effects can, and how that type of model is essential to Composobot's doing that thing it do.

3.1 Markov Chains

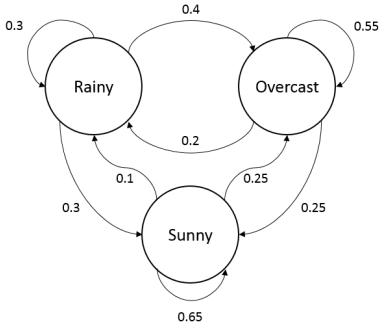
Markov chains are Markovian processes relying on the Markov assumption. As you may notice, that's not super helpful, because "Markov" is not a real word with a definition. All of this stuff is named for Andrey Markov, a mathematician whose privilege it was to discover and begin the codification of these ideas[10]. His name is fantastic and I love it, but this naming scheme does put us in the position of having to explicitly define some terminology that isn't very self-descriptive, which we'll jump straight into now.

Imagine some chain of events that happen in order, such as words in a sentence or the daily weather over the course of a week. "*The slithy toves did gyre and gimble in the wabe*", or *Sunny* followed by *Overcast* followed by *Rainy*. These chains of events could be thought of as series of "states". In the case of a sentence, each unique word can be thought of as the "state" that the sentence is currently in at some point in time. In the case of weather, *Sunny*, *Overcast*, and *Rainy* are "states" that the weather is in during some given day.

Now imagine that you want to know how likely it is that "*wabe*" appears as the final word in a sentence, or how likely it is that it will rain tomorrow. The Markov assumption is that this likelihood really only depends on some small number of states that come directly beforehand, rather than the entire text that precedes the sentence or the entire history of weather in a region. The most commonly made Markov assumption is called a first-order Markov assumption, which is that the likelihood depends on just the one preceding state[10].

If we're making the Markov assumption, then the chain of events in question is called a Markov chain. We can represent a Markov chain graphically, or by a matrix of the probabilities of transitioning from one state to another. The representations are called Markov transition graphs and Markov transition matrices, respectively. Taking weather as an example, a Markov transition graph and Markov transition matrix that describe such a Markov chain are given in figure 3.1 below.

Figure 3.1: Weather Example



	Sunny	Rainy	Overcast
Sunny	0.65	0.1	0.25
Rainy	0.3	0.3	0.4
Overcast	0.25	0.2	0.55

According to this model, the probability of the weather being *Rainy* tomorrow given that it is *Overcast* today is 0.2, while the probability of it being *Overcast* tomorrow

given that it is *Overcast* today is 0.55. You might imagine beginning in the node of the graph that represents today's weather and traveling along the arrows originating from that node once per day, arriving in the node that represents the next day's weather, repeating this process again and again until some unimaginable calamity brings about the end of all weather or until you get bored with the model.

This is, in essence, a Markov chain: a stochastic model that represents probabilistic transitioning from state to state dependent on the state or states that directly precede the transition. As simple as these models seem, they end up being very widely and usefully applicable; certainly they have been shown to be applicable in the domain of music, or we wouldn't be talking about them here[11][10][7][12][8][9]!

3.2 Hidden Markov Models

In the case of modeling weather, we enjoy the convenient advantage of knowing what the weather is like. We can look out the window and we just know. There's no need to stand outside with our eyes closed and infer the weather by whether we get wet or sunburned or just bored. We can observe the phenomenon we're interested in directly.

This isn't always the case! Suppose we did have to test whether we were wet or sunburned in order to know the weather, and deduced it was *Cloudy* only when neither of those two phenomena occurred. That's possibly difficult to imagine, because we can probably think of a few ways to troubleshoot that problem: open our eyes, or ask a friend, or fire a laser through the open air at a sensor to see what happens to the beam.

Let's imagine another wacky situation instead. Imagine that someone is sitting inside a shed and reading a book. She has three buttons in front of her, each corresponding to a different color: *Red*, *Blue*, and *Green*. You're sitting comfortably in a different shed, and you have three colored bulbs, of those same three colors, in front of you. Whenever the Reader reads a noun, she will press the *Red* button, and the *Red* bulb will light up in your shed. Whenever she reads a verb, she will press the *Blue* button, which will light up the corresponding bulb on your end. Whenever she reads a word that is neither, she will press *Green* and your *Green* bulb will be lit.

This is an example of a simple Hidden Markov Model: you can observe directly the transitions between colors, but you can't observe the transitions between words in the

Reader's text. The model is called "hidden" because the phenomena we want to model, in this case the words the Reader is reading, are hidden and cannot be observed directly. We can't directly observe the transitions between states, but only their "emissions", or their consequences[13].

Often, these emissions are even more convoluted than in our example. Suppose that the Reader was a little careless, and pressed the wrong button 5% of the time. Now our observations have to be weighted by our confidence that the color we see actually corresponds to the type of word the Reader is reading. Maybe the Reader gets very excited about verbs, and is even more careless when reading a verb, pressing the wrong button 10% of the time. This makes *Blue* even less likely to be the correct color whenever we see it, but also makes *Red* and *Green* slightly less likely! Then suppose that the wiring between the sheds is a little slapdash, and has a tendency to malfunction when it's raining. Now we have to look outside whenever we see a color, and consider the probability that the wrong bulb is lighting up depending on its color, and whether it's raining... things get complicated quickly. C'est la science.

The Markov models we deal with in this work won't be quite that messy, but they do model hidden information. We can observe straightforwardly the proportionality of the chord $CM7$ following Am, but sometimes progressions of chords depend on their position in some kind of abstract rhythmic structure, or on their proximity to the conclusion of a piece, and nobody fully understands that rhythmic dependency. We calculate based on the emissions, such as the probability of $CM7$ given that Am precedes it, and that $CM7$ will be the third chord in a repeating $1-2-3-4$ count, and that it is not too close to the end of the piece. We seek to capture as much information as we can about that abstract rhythm structure without being able to observe it directly. Consequently, most of the Markov chains in this work are in fact Hidden Markov Models.

If this section was abstract, dull, or difficult to follow, worry not. The important lessons to take away from this chapter are that Markov chains model transitions between states using conditional probabilities, and that sometimes it isn't possible to directly observe what we're modeling, and so we have to consider some extra variables to try to get at that information indirectly. If that makes sense, then congratulations! You've successfully learned things from this chapter and are well-equipped to move on to the good stuff!

Chapter 4

Anatomy of Composobot

The goal of this work is not just to understand Music Information Retrieval and how it might relate to Algorithmic Composition, but to actually build a program that walks the walk: preprocessing input for analysis, performing the information retrieval, and then composing original music based on what it learns.

As you may recall or guess, that program has been built, and their name is Composobot! The details of how Composobot works make up Chapters 5, 6, and 7. First, though, I'd like to give a brief overview of Composobot so that we have a rough idea of their scope and structure, a frame through which to view the finer details as we dive more deeply in.

Composobot's overarching goal is to take a corpus of musical pieces in MIDI format as input, learn patterns from them, and then compose original music based on what they have learned, outputting its compositions as ready-to-play MIDI files. Their process consists of three broad steps: preprocessing, learning, and composition.

The preprocessing step (Chapter 5) consists of reading in a list MIDI files, determining the key and mode of each piece individually, transposing the pieces to C Major or C Minor depending on whether the piece itself is Major or Minor, and then writing them out as new MIDI files that are labelled as Major or Minor.

The learning step (Chapter 6) is somewhat more involved. The preprocessed MIDI files are read in and separated into Major and Minor mode sets, as Major and Minor patterns will be learned and kept separately. Then each piece is separated into time divisions of one half measure, and the set of notes in each division is used to identify

that division's chord. This generates a sequential list of chords that form the chord progression for that piece, and the progressions for all pieces within a mode set are considered to build a Markov transition matrix for chord states, said states including a little extra information about the local and global positions of that chord within the piece.

Then, rhythm patterns are learned using the same half measure divisions by observing the rhythm of accompaniment notes in that half measure. A Markov transition matrix for accompaniment rhythm patterns is generated, conditional on preceding accompaniment rhythm pattern.

Finally, the melody voice is divided into "melodic phrases" using boundary detection on the melody voice. Markov transition matrices are generated for melodic phrase rhythm patterns conditional on preceding melodic phrase rhythm patterns. Markov transition matrices are also generated for the melodies within those phrases, conditional on previous melodic notes and on underlying chord.

Note that in all cases, rhythm patterns for Major and Minor mode sets are combined under the assumption that rhythm is not significantly conditional on mode.

Composobot's composition step (Chapter 7) consists of reading in the transition matrices and parameters from the learning step and then initiating a series of Markov processes to generate different elements of a new piece. Parameters are set at runtime for the key, mode, and length of the piece to be generated. Then, a chord progression is generated using the chord transition matrix for the appropriate mode (Major or Minor), and chord substitutions are made probabilistically based on a calculated "distance" between chords. A progression of accompaniment rhythm patterns is generated, and the chord progression given expression using those rhythm patterns. A progression of melody phrases is similarly generated from its transition matrix, and rhythms within those phrases are given note values using the melody transition matrix, note values being conditional on previous note within the phrase and on underlying chord. Finally, the piece is written out to a MIDI file that can be easily played on most systems.

Without doubt, that overview raises a lot of questions. Stick with me! It will be my earnestest enterprise to answer them in the coming chapters.

Chapter 5

Composobot: Implementation and Preprocessing

A program that learns how to write its own music sounds like a really great idea! Unfortunately, and to my utter dismay, in order to have written such a program, it is necessary at some point to actually write it. C'est la vie.

In this chapter we'll first examine how Composobot was, in fact, implemented, and what kind of input data they expect and receive. Then we'll look at the first graceful step in the syncopated waltz that is Composobot's process: preprocessing their input files to transpose them all to the key of C and separate them into sets of Major and Minor pieces.

5.1 Implementation

Composobot is implemented entirely in the Julia programming language, a dynamic programming language optimized for numerical computing[14]. The unabridged Julia source code for Composobot can be found in Appendix A, though I would inflict on nobody the task of attempting to fully understand the program solely by that code.

In addition to being both speedy and easy to use in itself, Julia offers the advantage of easy processing of MIDI data thanks to Joel Hobson's MIDI.jl package[15]. As discussed in section 2.1, MIDI note events are, by default, encoded with their literal timing in milliseconds. While ideal for accurately reconstructing a piece from data, this

representation entails an additional layer of challenges if we are interested primarily in the metrical timing of notes- that is whether they are, for example, quarter notes or half notes, and where they can be placed in a measure.

Hobson's MIDI.jl package structures MIDI data in a way very convenient for those purposes. The timing of a piece is divided into "ticks", with some number of ticks comprising a "beat". "Beat" here has the traditional meaning: in a piece in 4/4 time, for example, each measure consists of 4 beats, with each beat having the length of one quarter note. The number of ticks in a beat, then, describes the resolution of the timing in a piece. If a beat consists of 96 ticks, then a quarter note has a duration of 96 ticks; a sixteenth note has a duration of 24 ticks; a half note has a duration of 192 ticks.

A "note" object in MIDI.jl includes information about its duration and its position within the piece, both measured in ticks. Additionally, an object representing the MIDI file of a piece stores a value representing the number of ticks per beat for that piece. So, supposing we know the time signature of a piece, we can extract the metrical timings we want in the following way:

Let $\frac{a_P}{b_P}$ denote the time signature of a piece P, let t_P denote the ticks-per-beat of that piece, and let x_n and d_n denote the position and duration, respectively, of a note n in P.

$\delta(n)$, the metrical duration of n in terms of fraction of a measure, is given by

$$\delta(n) = \frac{d_n}{t_P b_P} \tag{5.1}$$

$\beta(n)$, the position of n within its measure, is given by

$$\beta(n) = \frac{x_n}{t_P a_P} \bmod 1 \tag{5.2}$$

And $\alpha(n)$, the measure in P that n falls within, is given by

$$\alpha(n) = \left\lfloor \frac{x_n}{t_P a_P} \right\rfloor + 1 \tag{5.3}$$

This allows for straightforward conversion from the data in the representation offered by MIDI.jl to the metrical timings we want to work with. Excellent!

The last crucial element of a note in MIDI.jl that we need to discuss is its representation of a note's pitch class and octave. This, too, is straightforward: a note's overall

pitch is represented by an integer value, with middle C ($C5$) being represented by the value 60. This means that we can easily extract a note's pitch class and octave with a little modulo arithmetic.

Let n again denote a note, and let v denote its integer value representation of its pitch class and octave. Let $F : V \to N$ where $V = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$ and $N = \{C, C\#, D, D\#, E, F, G, G\#, A, A\#, B\}$ be an ordered mapping from an integer value in V to a pitch class in N such that $F(v_i \in V) = n_i \in N$.

$p(n)$, the pitch class of n is given by

$$p(n) = F(v \bmod 12) \tag{5.4}$$

$o(n)$, the octave of n is given by

$$o(n) = \left\lfloor \frac{v}{12} \right\rfloor \tag{5.5}$$

MIDI.jl also provides architecture and a suite of useful functions for reading and writing MIDI files that, while of enormous utility to Composobot's functioning, are not conceptually relevant here. These and many other characteristics of MIDI.jl are best described by its documentation[15].

5.2 Corpus Construction

Before Composobot jumps right in and starts analyzing pieces, it needs a corpus of pieces to analyze!

Composobot doesn't have any idea how to gather a corpus of MIDI files, and so I'm not embarrassed to admit that the author shouldered most of the burden on this step. Sheet music for a selection of pieces of classical piano and related music was gathered with the goal of providing a relatively representative sample of the style of classical piano. The full list of pieces in the corpus can be found in Appendix C.

These pieces were then converted to MIDI files by hand using Tabit, a software designed for composition and playback of MIDI music using a tablature representation[16]. Small simplifications were made to these pieces: volume dynamics, modulation/vibrato, portamento, and other elements of performance expression were abstracted away. Additionally, the melody voice and accompaniment were encoded on separate tracks so

that they can be considered separately by later processes without needing to detect and separate them algorithmically. This seemed a reasonable simplification to maintain, since the voices are generally separated that way in the source sheet music.

Once the transcription is done, the files are gathered into a single folder and the list of their names is given to Composobot so that they know what to preprocess.

5.3 Key-Finding

Once the corpus is assembled, Composobot leaps into action! What they want to do is transpose each piece to C Major or C Minor, so that they can learn patterns for abstract "Major" and "Minor" patterns, rather than generating twelve different Major and twelve different Minor pattern sets. In order to accomplish that, they need to know what key each piece is in to begin with!

In pursuit of this lofty aim, Composobot reads each piece and compiles a probability distribution of the notes in that piece by direct observation. They compare this observed distribution to a set of key profiles derived from the Hu, Saul[6] profiles discussed in Section 2.2 using 12-dimensional Euclidean distance to determine a set of "key distances".

Figure 5.1: Exact Key Profiles

Key Profile	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
C Major	0.32	0.01	0.03	0.01	0.22	0.04	0.01	0.225	0.01	0.03	0.07	0.025
C Minor	0.17	0.01	0.13	0.17	0.01	0.12	0.01	0.16	0.1	0.02	0.03	0.07

Let the observed probabilities in a piece P be denoted by $O_P = \{o_1, o_2, \cdots, o_{12}\}$, and let the twenty-four key profiles derived from the Hu, Saul profiles be denoted by $K_j = \{k_{j_1}, k_{j_2}, \cdots, k_{j_{12}}\}$ for $1 \leq j \leq 24$. Then the "key distance" $d_k(O_P, K_j)$ between the observed distribution and a given key profile is given by

$$d_k(O_P, K_j) = \sqrt{(o_1 - k_{j_1})^2 + (o_2 - k_{j_2})^2 + \cdots + (o_{12} - k_{j_{12}})^2} \tag{5.6}$$

Generally, the key profile with the smallest distance from the observed distribution should be considered the key of the piece. However, this is complicated by the fact that each Major key has a relative Minor, and vice versa. A key and its relative key have the

same notes in their scales, though they have different tonics, or root notes. For example, the C Major scale consists of all the white keys on a piano; so does the A Minor scale. This can make it difficult to differentiate between relative keys by distribution alone.

To help alleviate this problem, Composobot weights the distance calculation such that $d_k(O_P, K_j)$ is artificially lower if K_j represents the key of the piece's first measure. For K_t denoting the key profile matching this favored target, the weighted distance $d'_k(O_P, K_t)$ is given by

$$d'_k(O_P, K_t) = d_k(O_P, K_t) - 0.1 \tag{5.7}$$

To find this favored target key profile, Composobot first determines the key of the piece's first measure using Equation 5.6, with the modification that O_P in that calculation represents only the observed distribution of the first measure. The least distant key profile becomes the favored target key profile, such that when Equation 5.6 is applied to calculate distances for the whole piece, Equation 5.7 is substituted when calculating the distance from the favored target profile. The least distant key profile in this calculation is considered the key of the piece.

5.4 Transposition and Labeling

Once Composobot is fairly certain it knows a piece's key, they want to transpose that piece up or down to the nearest octave of C. This is done by arithmetic transformation on the pitch value of each note in that piece: if the piece is found to be in the key of A, for example, then the pitch value of each note in the piece is incremented by exactly 3, the distance between C and A. Alternatively, if the key of the piece is found to be F, each pitch value is decremented by exactly 5, since it requires fewer incremental steps to transpose down to F from C than it does to transpose up. If the key of a piece is found to be C then no transposition happens at all.

Once this transposition is done, Composobot writes the piece back out to a new MIDI file which is labeled in its filename as being either Major or Minor. This labeling is done uniformly so that when Composobot reads a file during its learning step, they can recognize from the filename whether the piece should be considered Major or Minor. At the triumphant conclusion of this step, Composobot has ensured that the corpus is

divided into Major and Minor partitions to be analyzed separately, and that all pieces in each partition are in the same key so that each partition can be analyzed all together.

At this point, Composobot is ready to analyze our corpus and learn how music works!

Chapter 6

Composobot: Learning

At this point, we have a solid corpus of Major and Minor pieces, all transposed to the corresponding key of C and labeled with their mode. Time for the good stuff: reading the music and learning all about it!

Composobot is going to look at the pieces in two different ways: in terms of a piece's chords and chord progression, and in terms of the melody voice. It will start by determining and labeling chords, build those into progressions, and then estimate Bayesian probabilities of substituting chords with other chords based on a measure of distance between the chords. The characterization of chords used by Composobot, while relying on previous work in the field, differs significantly from those used by other systems and will be detailed in section 6.1 below.

Chord progressions are used to construct a Markov transition matrix for each mode (Major and Minor). Markov states for this matrix consist of the current chord, the previous chord, and measures of local and global position of the chord within the piece.

The accompaniment of pieces is then analyzed in terms of its rhythm, and a "rhythm pattern" is built for each half-measure that represents the rhythm of the accompaniment during that period. These are then built into progressions in the same way that chords were, and a Markov transition matrix is similarly constructed.

Once Composobot has analyzed the accompaniment of a piece, it will move on to the melody. Melody will be divided into melodic phrases using a measure of boundary detection tailored to tonal melody. These phrases are built into progressions and then analyzed in two very different ways. In the first, each phrase is generalized to a "rhythm

only" representation that abstracts away note values and overall position within a piece, and these progressions of phrase rhythms are used to calculated a Markov transition matrix for melodic phrase rhythm patterns. In the second, the pitch classes (e.g., $C5$ or $A\#6$) within each phrase are analyzed independently of rhythm, and Markov transition matrices are constructed for each mode that model the likelihood of a note given the previous note and the underlying chord at the time.

Finally, all of these Markov transition matrices are formatted and written out to a text file that can be said to contain the model that Composobot has learned from the pieces it has observed, and which can subsequently be read by Composobot and used to compose novel music during the composition step.

6.1 Chord-Labeling

The first step in chord-labeling is to determine how chords will be characterized. Composobot's representation of chords builds on previous work in the field and, I hope, finds an ideal middle ground that incorporates the advantages of several different methods while covering some of their disadvantages.

In section 2.3 we discussed the symbolic aggregate chord characterization (e.g., Am or $Caug7$) common in musical notation and the literal characterization (e.g., $a5c6e7$) used by Paiement, Eck, and Bengio[7]. Composobot seeks a compromise between the two extremes by beginning with a small knowledge base of predefined chords from the symbolic aggregate characterization, dividing pieces into slices of half-measure length, generating a literal characterization for each slice from the notes in that slice, determining which of the knowledge base chords the literal chord is "closest" to by 12-dimensional Euclidean distance, and then removing from the literal characterization all notes that do not correspond in pitch class to a note in its corresponding knowledge base chord.

Composobot begins with a knowledge base consisting of 60 chords: the 12 Major triads, the 12 Minor triads, the 12 Major seventh and 12 Minor seventh chords, and the 12 dominant seventh chords. These chords are represented within Composobot by 12-dimensional binary vectors where each dimension is a pitch class: the chord $Cdom7$, for example, is represented by the vector $\{1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0\}$ which corresponds to the notes $CEGA\#$.

The input piece is divided into time divisions with a length of one half-measure. In most cases, a literal chord representation for that division is constructed straightforwardly by observing each note that occurs within it. However, in the event that fewer than three notes occur within a division, that division will instead be labeled with the chord of the division that precedes it.

Literal chord representations taken from input data need to be compared to the 12-dimensional knowledge base chord vectors such that a distance can be defined between them. This requires that observed chords be translated into 12-dimensional pitch class vectors. That makes this a good time to define the concept of a note's "loudness" as adapted from the work of Paiement, Eck, and Bengio[7]. They observe that lower register notes are more impactful on a listener's perception of a chord than higher notes, and that lower notes of the same pitch class tend to have a masking effect on higher notes of that pitch class.

"Loudness" here has nothing to do with volume. Rather, it's a measure of a note's impact on the perception of the chord it belongs to. For a note n, the pitch class $p(n)$ of n as defined in Equation 5.4, the octave $o(n)$ of n as defined in Equation 5.5, and ρ a constant such that $0 < \rho \leq 1$, the loudness $l(n)$ of n is defined as

$$l(n) = \rho^{o(n)} \tag{6.1}$$

For Composobot's calculation of loudness, ρ is set at 0.99 by empirical determination. This means that the differences in loudness between different octaves is considered very small, but sufficient to break ties.

In the 12-dimensional vector representation of an observed chord, we want the loudness of the "loudest" note of each pitch class to be taken as that chord's value for that pitch class. For an m-note chord $X = \{n_1, \cdots, n_m\}$, the contribution x_i for pitch class i is defined as

$$x_i = max(l(n), n \in \{X | p(n) = i\}) \tag{6.2}$$

Then the 12-dimensional vector representation $v(X)$ of the chord X is given by

$$v(X) = \{x_1, x_2, \cdots, x_{12}\} \tag{6.3}$$

Finally, the distance $d(X, K_i)$ between the observed chord X and the ith knowledge base chord $K_i = \{k_{i_1}, k_{i_2}, \cdots, k_{i_12}\}$ is given by

$$d(X, K_i) = \sqrt{(x_1 - k_{i_1})^2 + (x_2 - k_{i_2})^2 + \cdots + (x_{12} - k_{i_{12}})^2} \tag{6.4}$$

Once the nearest knowledge base chord K_t has been identified, the chord X is reduced to X' such that each note in X' has pitch class corresponding to a pitch class represented by a 1 in K_t.

For example, let X be the literal representation chord $c4g4c5d5e5f5g5b5$, which has pitch classes $\{C, D, E, F, G, B\}$. This chord would be found to be nearest to the knowledge base chord $\{1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1\}$, which has pitch classes $\{C, E, G, B\}$. Then the reduction step would remove all notes in X with pitch classes in $\{D, F\}$, the difference of the two pitch class sets. Removing $d5f5$ from X, then, produces $X' = c4g4c5e5g5b5$.

In the event that, like in the above example, the resulting chord contains more than five notes, the notes with the least loudness- that is, the highest notes- are removed until only five notes remain. This is done to ensure that observed chords of a certain high degree of similarity are considered together, and to limit the size of the space of observed chords. This threshold was determined empirically to optimize capturing similarity without discarding too much information.

Finally, it's important to consider the case where the removal step results in there being just one or two notes in X', which is less than ideal for defining a whole chord. In the event that $|X'| < 3$, Composobot will add in notes from the nearest knowledge base chord K_t until $|X'| = 3$ by iteratively adding in notes of the lowest pitch class in K_t not present in X'. Notes are added in the fifth octave of their pitch class for simplicity.

This process, which occupies some middle ground between a restricted symbolic chord representation and a literal representation, helps to address the shortcomings of both: the space of possible chords is much greater than if it were limited to a relatively small set of symbolic representations that abstract away octave, the possible chords capture varied voicings, and the space of possible chords is still much smaller than an unrestricted literal representation.

6.2 Chord Progressions

Given that we've now sliced up our pieces and labeled the slices with chords, there is a very easy step we could take to build a chord progression: simply list the names of the chords in sequence! However, it may not be the worst idea to take a moment to consider how we're going to use this progression, what shortcomings that simple list might have, and how we might get around them.

For that reason, we're going to zoom out and consider the composition step that we hope this information retrieval process will inform. The bones of the composition step will all be Markov processes, and certainly the composition of chord progressions will begin with a Markov process of some kind on chords. We could construct a Markov transition matrix that could generate a new chord progression, then, by just putting our chord labels in a sequential list. Conditional probabilities of a chord following another chord in sequence could be calculated by observation on these lists. If we do it right, we should end up with chord transitions that follow smoothly from one another.

If we stop to think about any of our favorite pieces, however, it may become clear that a general Markov assumption, that a chord depends only on the chord that precedes it, is going to be a poor one to make here. There is some broader, more global structure that informs the choice of the next chord in a progression. In particular, it is common to see a progression broken up into "chord phrases": progressions of, say, 4 or 8 chords that repeat throughout a piece. Additionally, we may observe that the end of piece generally sounds different than the middle of a piece; a final "chord phrase" is more likely to bring some resolution to a piece than a phrase occurring at some arbitrary place in the body of the piece.

This implies two pieces of information we would like our system of Markov dependencies to know about the chords it is considering. First, it should consider what I will call a chord's local cyclic position: its position within a "chord phrase", as defined by dividing the progression sequence into groups of four chords and labeling each chord as occupying position $1, 2, 3$, or 4 within its group.

The second piece of information it should consider is some measure of the chord's location in the global structure of the progression. In this case, we consider whether it falls within the final quartet of chords in the progression: a boolean flag that is true if

a chord is one of the last four, and false otherwise.

Given this, let's define our Markov states explicitly, and as consisting of more than just chord labels. For a Markov state M constructed at position i in a progression, let X_i denote the current chord at position i, X_{i-1} denote the previous chord in the progression, $\phi \in \{1, 2, 3, 4\}$ denote the local cyclic position of X_i, and $\omega \in \{0, 1\}$ denote whether X_i is one of the final four chords of a piece. Then M is defined as

$$M = \{X_i, X_{i-1}, \phi, \omega\} \tag{6.5}$$

Note that when $i = 1$, X_i will be the first chord in a progression and the chord at position $i - 1$ does not exist. In this case, X_{i-1} will be the symbol S representing the beginning of a progression. All analyzed and generated progressions, then, will begin with Markov states where $X_{i-1} = S$.

The progressions that are converted to Markov transition matrices for chords, then, are calculated directly from these progressions of Markov states.

6.3 Chord Substitution Probabilities

One drawback of any kind of literal chord representation, as we have seen that Composobot employs, is that the space of possible chords is vast, which makes it difficult to gather sufficient observations of each chord. This is only compounded by our considering chord progressions as Markov chains. There is likely to be at least one chord that is seen only once in the corpus, which makes it impossible to directly calculate a conditional probability of that chord following each other observed chord- for all but one of those cases, we never saw that succession happen!

Paiement, Eck, and Bengio continue to exert their due share of influence over Composobot's chord modeling by concocting an ingenious solution to this problem. Rather than trying to impute conditional probabilities via some kind of smoothing, their model considers once again the Euclidean distances between chords and uses those distances to estimate probabilities of substituting one chord for another within a progression[7].

For chords X_i and X_j in a set of s chords, $d(X_i, X_j)$ as defined by Equation 6.4, and the parameter λ such that $0 \leq \lambda < \infty$, the probability $P_{i,j}$ of substituting X_i for X_j in

a progression is given by

$$P_{i,j} = \frac{e^{-\lambda d(X_i, X_j)}}{\sum_{j=1}^{s} e^{-\lambda d(X_i, X_j)}} \tag{6.6}$$

The parameter λ controls the likelihood of substitution overall. When $\lambda = 0$, all chords have an equal probability of being played at any time, since the probability of substitution is uniform for each pair of chords (including a chord and itself!). As λ approaches ∞, the probability of any given substitution approaches 0, and no substitutions are made at all. For Composobot's calculation of substitution probabilities, λ is set at 5 by empirical determination.

In order to reduce computation and storage, probabilities are calculated only for pairs of chords X_i, X_j where $d(X_i, X_j)$ is less than some threshold, set at 1.2 by empirical determination. This prunes only substitutions that would be exceedingly rare, but reduces multiplicatively the computational demands of calculating probabilities and of later making substitutions.

Probabilities are calculated for each such pair of chords once all pieces have been chord labeled. Note that this includes the probability of substituting a chord with itself- for most values of λ, this probability will dominate the substitution probability space. This calculation can be computationally demanding depending on the number of chords observed, but only needs to be performed once per model. The use of these probabilities in making substitutions occurs during the composition step, which is described in Chapter 7.

The use of substitution probabilities means that every time we observe any chord in our corpus, we are also observing every other chord in the corpus to the extent described by the probability of substitution between the two. We don't need to worry about calculating every exact conditional probability for some rare chord given any other chord preceding it, since any chord that succeeds the preceding chord may probabilistically transform into the rare chord.

6.4 Accompaniment Rhythm Patterns

A chord progression suggests which notes are be more likely to be played during each time division, but it suggests nothing about *when* those notes appear *during* a time division. Are all of the notes played at the same time? Are they sounded once and held, or are they a rapid succession of sixteenth notes? Are there rests, pauses? And does any of that depend on the rhythm of the notes in previous time divisions?

To get at this, we will build a Markov transition matrix similar to the one we built for chord progressions, but for the rhythm patterns within time divisions. Rather than chords, we will model the progression of rhythm patterns, which are constructed in the following way.

First, a time division of one-half measure is divided into eight equal portions, each one-sixteenth note in length. Then an eight-digit string of zeroes is generated. For each of the eight portions of the time division, we check whether a note's onset falls within that portion- if it does, we check its duration. This duration is converted into a number of sixteenth notes (e.g., a quarter note is four sixteenth notes, and is converted to the value 4). Then, this value replaces the zero in the generated string at the index matching its portion.

For example, suppose we have a time division containing a quarter note followed by two eighth notes such that none of these notes overlap. Then the quarter note at the beginning of the division is converted to value 4 at position 1; the quarter note halfway through the eight-portion division is converted to value 2 at position 5; the final quarter note is converted to value 2 at position 7. The rhythm pattern for this division would be the string "40002020".

This generated string not only describes the rhythm of the time division in a reconstructible way, but also serves as the rhythm label for that division in the same way that $Am7$ might serve as the chord label for a division.

The Markov transition matrix for rhythm patterns is in most ways identical to the one we built for chord progressions. Rather than chords as the first two arguments of a state it will use rhythm patterns, but states will again include a ϕ measure of local cyclic position and an ω measure of global position. Once again, an abstract start-of-piece symbol S will serve as the "previous rhythm pattern" for the first state in a progression.

Explicitly, for a Markov state M constructed at position i in a progression, let $\mathcal{R}_i$ denote the current rhythm pattern at position i, $\mathcal{R}_{i-1}$ denote the previous rhythm pattern in the progression, $\phi \in \{1, 2, 3, 4\}$ denote the local cyclic position of $\mathcal{R}_i$, and $\omega \in \{0, 1\}$ denote whether $\mathcal{R}_i$ is one of the final four rhythm patterns of a piece. Then M is defined as

$$M = \{\mathcal{R}_i, \mathcal{R}_{i-1}, \phi, \omega\} \tag{6.7}$$

The Markov transition matrix for rhythm patterns is calculated directly from the progression of these states.

6.5 Melodic Phrase Boundary Detection

The problem of appropriately detecting the boundaries between melodic phrases is not trivial. However, as suggested in Section 2.4, we're going to get a hand from the work of David Temperley, who devised a system of what he calls Phrase Structure Preference Rules (PSPR) for determining the boundaries between melodic phrases[9]. In particular, I adapt the three rules he calls the Gap Rule, the Phrase Length Rule, and the Metrical Parallelism Rule and implement them in Composobot to divide a melody into phrases.

The Gap Rule ($PSPR_1$) draws on the intuition that boundaries between melodic phrases are more likely to occur at longer gaps between the notes in a melody. Recall that the inter-onset interval (IOI) between two notes is the duration between the start of one note and the start of the following note, and that the offset-to-onset interval (OOI) is the duration between the end of one note and the start of the next. Then for two successive notes n_i, n_{i+1} in a melody, their $PSPR_1$ score is defined as

$$PSPR_1 = \frac{IOI(n_i, n_{i+1}) + OOI(n_i, n_{i+1})}{\overline{IOI}} \tag{6.8}$$

$\overline{IOI}$ here denotes the mean IOI for all pairs of successive notes in the melody.

The Phrase Length Rule ($PSPR_2$) is derived from the observation that most melodic phrases tend to be between 6 and 10 notes in length. $PSPR_2$ scores for notes in a melody are calculated iteratively as phrase boundaries are drawn, so that a note's $PSPR_2$ score is based on its index in a list that begins with the first note in the piece not already

part of a melodic phrase. Seeking to penalize phrase boundaries more the further they are from the 6 to 10 note range, and letting n denote the index of a note in the sequence of notes note already in a melodic phrase, a note's $PSPR_2$ score is defined as

$$PSPR_2 = -|log_2(n) - 3| \tag{6.9}$$

The difference is calculated from 3 because $3 = log_2(8)$, and 8 is equidistant from 6 and 10. The score is negative because a larger distance makes it less likely that a phrase boundary should be drawn at that note.

The Metrical Parallelism Rule ($PSPR_3$) relies on the assumption that melodic phrases are more likely to begin on stronger beats of a measure, and more likely to begin on the same beats as one another. For our purpose, it was observed in the corpus that most melodic phrases begin exactly at the first beat of a four beat measure, and so the $PSPR_3$ score seeks to penalize notes more the further they are from the first beat of the measure. Dividing measures into increments of 16 beats and letting $p(n)$ denote a note's position in a piece in 16 beat increments, a note's $PSPR_3$ score is defined as

$$PSPR_3 = -log_2((p(n) - 1) mod 16 + 0.99) \tag{6.10}$$

The addition of 0.99 to the argument prevents logarithms of zero. The score, as with $PSPR_2$, is negative because a note is less likely to be a phrase boundary the later in its measure it appears.

A note n_i's overall $PSPR$ score is simply the sum of these rule scores.

$$PSPR(n_i) = PSPR_1(n_{i-1}, n_i) + PSPR_2(n_i) + PSPR_3(n_i) \tag{6.11}$$

Composobot divides a melody into phrases by first building a list of all notes in order. It then calculates the $PSPR$ score of every note in that list, finds the note with the highest $PSPR$ score, and draws the phrase boundary before that note. It stores every note before the boundary as a phrase and then repeats the process, taking as its new list the list of all notes not already assigned to a phrase. It iterates through the melody this way until each note has been assigned to a melodic phrase.

At this point, Composobot will have melodic phrase progression, an ordered list of melodic phrases that appear in a piece. Doing this with every piece in the corpus gives

us a series of progressions, and it is from this set of observed melodic phrase progressions that Composobot will learn what it can about how to build melodies.

6.6 Melodic Phrases: Rhythm

When we were learning from chord progressions, we discussed the relative advantages of different ways of representing chords. Too general a representation and we have a nice small set of states but a lot of lost information; too specific a representation and we can capture dependencies very accurately, but the number of possible states means the amount of dependencies we're trying to capture is too large for any corpus we have available. We sought a middle ground that tried to retain the most important information without over-bloating the number of possible states.

Here, we must be even more careful. If we try to capture the conditional probability of one literal melody phrase following another, we're going to end up with an enormous space and sparse observations within it: the number of possible melody phrases is astronomical, and to calculate accurate probabilities of any of them following any other would require a number of observations that is almost certainly larger than the number of musical pieces humanity has produced. Needless to say, our corpus is somewhat smaller than that.

Instead, we will model melody on two levels. First, we will convert every melodic phrase to a rhythm pattern representation: a series of onset positions and durations that describe the rhythm of the melodic phrase relative to its own beginning. For example, a melodic phrase consisting of the pitch-duration-position tuples {($C5$, 96, 768), ($D5$, 96, 864), ($B4$, 48, 933), ($C5$, 144, 981)} would be converted to the duration-position rhythm pattern representation {(96, 0), (96, 96), (48, 192), (144, 240)}. The pitch classes are abstracted away, the durations remain unchanged, and the positions are all decremented by the value of the first position in the phrase, leaving all positions relative the start of the phrase.

Once each phrase is converted, we analyze progressions of rhythm-pattern phrases and calculate conditional probabilities between them. Probabilities are calculated for the first phrase in each progression given an abstract "start of piece" symbol, and probabilities are thereafter calculated for a phrase given the phrase that precedes it. It

is assumed that many melodic phrases that vary in terms of the pitch classes of their notes will be the same when analyzed in terms of rhythm alone, and this assumption is confirmed empirically.

The learning for these rhythm-pattern phrase probabilities is performed on combined Major and Minor progressions under the assumption that the conditional probabilities for rhythm patterns alone are independent of mode, which has been supported empirically.

6.7 Melodic Phrases: Pitch

The pitch classes of a melody can't be ignored altogether, of course. As discussed previously, though, we need to make compromises of some kind in order to keep our state space from outpacing our corpus. To that end, we make one of our more dubious assumptions: that the pitch class of notes in a melodic phrase are independent of the rhythmic structure of that phrase. That is, that the specific rhythm pattern of a melodic phrase has no influence on the progression of the notes within that phrase.

Specifically, the pitch class of a note in a melodic phrase will be assumed to be dependent on the pitch class of the note preceding it and on the underlying chord at the point of its onset. We return to considering our larger set of observed melodic phrase progressions, and not the rhythm form representation of those progressions. Then, for each phrase in a progression, the conditional probability of the first note's pitch class given an abstract "start of phrase" symbol and the underlying chord at its onset position is calculated, and thereafter, we calculate the conditional probability of a note's pitch class given the previous note's pitch class and the underlying chord and the current note's onset position. Probabilities would take the form of $P(C5 \mid \text{“Start”}, \text{“}Am7\text{”})$ or $P(A6 \mid C5, \text{“}CM\text{”})$ where $C5$ and $A6$ are pitch classes, "Start" is the abstract "start of phrase" symbol, and "$Am7$" and "CM" are the chords at the position where $C5$ and $A6$, respectively, begin.

Note that these conditional probabilities are calculated and stored separately for Major and Minor modes, following the same assumption that led us to separate Major and Minor modes for chords: that pitch classes are heavily dependent on the mode of a piece, and that pieces of different modes behave very differently when it comes to pitch

class.

6.8 Model Preparation and Representation

At this point, Composobot has constructed a whole mess of matrices, and it would behoove both Composobot and we the readers to pause and try to make sense of it all.

We've constructed two Markov transition matrices for chord progressions, one for each mode (Major and Minor), where each state contains the current chord, previous chord, local cyclic position of the current chord, and global position of the current chord.

We also constructed a matrix representing the probabilities of substituting any given chord for any other chord.

We produced one Markov transition matrix for accompaniment rhythm patterns, and one Markov transition matrix for melody phrase rhythm patterns.

Finally, we produced two Markov transition matrices for melody pitch values, one for each mode, where each pitch value depends on the preceding pitch and the chord underlying the current pitch, for a total of seven matrices or five types of matrices.

Figure 6.1: Model Matrices

Matrix	Description
Chord Progressions (M)	Markov transition matrix of chords in C Major
Chord Progressions (m)	Markov transition matrix of chords in C Minor
Chord Substitutions	Probability distributions of substituting one chord for another
Accompaniment Rhythm Patterns	Markov transition matrix of accompaniment rhythm patterns
Melody Rhythm Patterns	Markov transition matrix of melody rhythm patterns
Melody Pitch (M)	Markov transition matrix of melody pitch values in C Major
Melody Pitch (m)	Markov transition matrix of melody pitch values in C Minor

These seven matrices are written out to a text file using their literal representation in Julia code, one matrix per line. Composobot will then be able to use any such model

for composition by specifying the name of the text file, reading it in, and storing each line as a matrix, since the line is represented in Julia code. This intermediate step, storing a model as a text file, allows Composobot to learn a model only once and then compose with it as many times as they like!

Now that the model has been learned, we're ready to move on to fun part: algorithmic composition! In the next chapter we'll see how Composobot, as if by magic, uses the model they've learned to compose a novel musical piece!

Chapter 7

Composobot: Composition

We've assembled a corpus, preprocessed it, and even learned a model from it. No mean feat, that, and nobody could blame Composobot for being just a little bit tuckered out. Fortunately, Composobot lives for the glory of triumph and would not hesitate to choose death over defeat, unwavering in their zeal for seeing tasks through to completion. They are, in fact, positively champing at the bit to begin their climactic third act: algorithmic composition of novel music based on the model they have learned!

In this chapter we'll explore the details of that process step by step. We'll begin by discussing some of the parameter selection that will inform the final form of the composition, and then proceed through the mechanics of composition in a manner mirroring that of our exploration of the learning process. We will be looking at how each of the five types of matrices we wrote out to a model in Chapter 6 is used, sequentially, to probabilistically build a composition.

7.1 Parameter Selection

Before Composobot gets started on actually putting their novel musical piece together, there are few decisions that need to be made, a few parameters that need to be set that are not based on the patterns Composobot has learned. These decisions are relatively minor and could be made stochastically, but Composobot values the autonomy of humans above all else and so elects to solicit the preferences of their users.

These composition parameters are: the desired mode, transposition offset, and

length of the piece to be composed; the name of the model file to be used; a name for the final MIDI file that will contain the composed piece.

Figure 7.1: Composition Parameters

Parameter	Description
Mode	Major or Minor
Transposition Offset	Specifies key by number of half-steps to transpose up (positive) or down (negative)
Length	Desired length of piece in number of measures
Model Filename	Name of model file generated during learning
Output Filename	Desired name of output MIDI file

"Mode" specifies whether the composition should be Major or Minor, and is entered as a string. "Transposition Offset" is entered as a positive or negative integer, and the pitch of every note in the piece will be transposed by exactly that value (from the key of C). "Length" sets the length of the composition in number of measures in $\frac{4}{4}$ time and is a positive integer. "Model Filename" takes the literal name of the model file generated at the conclusion of the learning step- note that this allows the storage of any number of model files that be called at will to produce compositions. Finally, "Output Filename" is the desired file name for the MIDI file that the new composition will be written to.

Once these parameters have been selected, Composobot is ready to take the reins and ride: it's time to compose some music!

7.2 Chord Progression

Assuming that all parameters described in the previous section have been chosen, Composobot will proceed to load the model specified by reading in the raw text of the model file and then splitting it into seven matrices by the "newline" character, the normally hidden character in a text file that specifies when to start a new line of text. These seven matrices are represented literally in the text file by their Julia code, and so can be read into memory straightforwardly.

Depending on the mode (Major or Minor) that has been selected, Composobot will

load the Chord Progression Markov transition matrix for that mode. This matrix, as described previously, represents chords as "chord states". This representation is detailed in Section 6.2 and described by Equation 6.5- briefly, a state M_i consists of the current chord (X_i), previous chord (X_{i-1}), local position in a repeating 4-cycle ($\phi \in \{1, 2, 3, 4\}$), and global position ($\omega \in \{0, 1\}$) in terms of whether it is near the end of a piece.

The progression matrix contains conditional probabilities of selecting chord states given the chord state that was selected previously: $P(M_i|M_{i-1})$. Some subset of these probabilities take the form $P(M_i|S)$, where S is an abstract "Start State" and $P(M_i|S)$ represents the observed probabilities of a chord state appearing as the very first chord state in a progression.

To generate the first chord in the progression, Composobot samples from the $P(M_i|S)$ probabilities according to their distribution. Some chord state will be selected, and its corresponding chord will be the first chord in the generated progression. Thereafter, each new chord state will be selected by sampling from the distribution of chord states observed to succeed the previously selected state.

Let l represent the desired number of chords in the piece, defined as twice the value of the desired length of the piece in measures (chords occupy a space of one half measure). Given some already selected chord state $M_i = \{X_i, X_{i-1}, \phi, \omega\}$, the value $\phi' = ((\phi + 1) mod 4) + 1$, the value $\omega' = 1$ if $l - (i + 1) \leq 4$ and 0 otherwise, then M_{i+1} is constructed by sampling from the distribution of the following probabilities over all chords X_j: $P(X_j|X_i, \phi', \omega')$. The selected chord is denoted X_{i+1}, and $M_{i+1} = \{X_{i+1}, X_i, \phi', \omega'\}$.

The generated chord progression is an ordered list of chords so selected. Again, l is the desired length of the piece in number of chords: the generated chord progression is given by $\mathcal{P} = \{X_1, \cdots, X_l\}$. The other values of the chord states (e.g., ϕ and ω) are used only to generate these chords, and are not themselves stored after the ordered chord progression is generated.

7.3 Chord Substitution

Composobot's soul is a dancing star: now that a chord progression has been generated, Composobot longs to add a measure of beautiful chaos to the mix. Using the matrix of

substitution probabilities read in from the model, Composobot will consider each chord X_i in the generated progression $\mathcal{P}$ one by one and probabilistically substitute it with another chord. These probabilities were generated using the "distance" between chords, as described in section 6.3.

For each chord in the generated progression, Composobot will take a single random sample according to that chord's distribution of substitution probabilities, and replace the initial chord with the new, substituted chord. As mentioned in section 6.3, the probability of a chord being substituted with itself will usually dominate the probability space of its substitutions, with the result that most chords in the progression will not be observably changed by this substitution.

For example, consider a chord $X_i \in \mathcal{P}$. Suppose for simplicity that only four chords were observed in a corpus- a preposterously low number suggested only for example. Call them $\{X_1, X_2, X_3, X_4\}$, and suppose $X_i = X_1$. Suppose that X_1 and X_2 are pretty similar chords: CM and $CM7$, maybe. X_3 and X_4, on the other hand, are very dissimilar to X_1: $F\#m$ or $A\#9$ or something. Then the substitution probability vector for X_1 might look something like $\{0.91, 0.083, 0.003, 0.004\}$, where the jth position in the substitution vector represents the probability that X_1 will be replaced by X_j. Very probably, then, Composobot would replace $X_i = X_1$ with X_1 and no observable change would take place. However, it may reasonably replace $X_i = X_1$ with X_2, which does result in an observable change.

Note that this substitution is based entirely on a calculated "distance" between the chords, and not on any observed behavior in the corpus of analyzed pieces. This means that, after substitution, some X_k might follow some X_j in the generated chord progression, despite X_k never being observed to follow X_j in any of the pieces in the corpus. As discussed in Section 6.3, this allows us to capture a much larger share of musical patterns without requiring a correspondingly enormous corpus of musical pieces to analyze. It also allows us to occasionally be surprised by beautifully chaotic and limited dissonance!

Composobot will make exactly one pass through the chord progression, probabilistically substituting every chord. Once this has been done, the chord progression is final, and it's time to translate that abstract progression into an actual series of notes!

7.4 Accompaniment: Rhythm and Expression

Recall that in an earlier chapter we talked about chords acting as "central polarities" for the notes in a time division, rather than an explicit blueprint describing which notes will be sounded and when. Clearly, though, if Composobot is going to actually compose music they can play, they're going to need to make some "which" and "when" decisions!

The question of when notes will be played is answered by the Accompaniment Rhythm Patterns matrix in our model. Through a similar Markov process to that used to generate a chord progression, Composobot generates a rhythm pattern progression that covers the entire specified length of the piece being composed, and the patterns in that progression describe the rhythm of the notes to be played during their respective time divisions.

The question of which notes will be played is answered in a relatively straightforward and probabilistic way. As Temperley points out, the tonic, or root note, of a chord is more likely to fall on stronger and even beats of a measure: more likely to occur at whole note intervals than half, more likely at half than quarter, etc. Thirds and fifths, similarly, are more likely to fall on strong even beats[9]. As no one would blame you for having forgotten, chords were pruned to a size of at most five notes during the learning process. Accompaniment notes can therefore be selected from a set of notes whose size is known to be at most 5, and can be selected probabilistically based on the position of that note within its time division.

Composobot does just this. Given a rhythm pattern in the form of an eight-digit string, a note is selected for each entry according to four different distributions: one for notes in position 1, one for notes in position 5, another for notes in positions 3 and 7, and a fourth for notes in the remaining positions of 2, 4, 6, and 8. The distributions choose notes according to their "loudness", or impactfulness in the identity of the chord, as measured by Equation 6.1. The most impactful notes are those more likely to fall on strong even beats, and are more likely to be selected by the earlier distributions than the later ones. The least impactful notes, conversely, are more likely to be selected by the later distributions than the earlier. It is worth noting that at each position, the number of notes to be played is also probabilistically determined according to the same distribution- at least one note will always be selected, but as many as three notes might

be.

The execution of this step proceeds like so: using the Accompaniment Rhythm Patterns matrix from the model, Composobot generates a rhythm pattern progression equal in length to the chord progression that has already been generated. Let this length be denoted l; then for each $i \in \{1, \cdots, l\}$, Composobot generates notes according to rhythm pattern $\mathcal{R}_i$ using notes from the chord X_i, and adds those notes, in order, to a blank MIDI track.

Once this is done, we have a MIDI track containing our whole accompaniment! We could export it right now and listen to it, but let's treat ourselves by patiently maintaining our anticipation while Composobot generates a melody for these notes to accompany.

7.5 Melodic Phrase Progression

Melody is generated in a way analogous to the generation of accompaniment we've already seen. We'll start by generating a progression of melody phrase rhythm patterns using the Melody Phrase Rhythm Pattern matrix from our model. The biggest difference in the processes is that we don't know *apriori* how many melody phrase rhythm patterns to generate!

Recall that chords in the chord progression had a length of exactly one half measure. If we are generating a 16 measure piece, for example, then we know that we need to generate exactly 32 chords in progression, and we will have covered the entire piece. Melody phrase rhythm patterns, on the other hand, do not have a uniform length. Their length was determined according to the Phrase Structure Preference Rules detailed in Section 6.5, which detected and defined boundaries according to rhythmic gaps between notes, the number of notes in a phrase so far, and the metric location of the possible boundary. Not only do none of these rules translate directly into a metric length, but these rules weight possible boundaries probabilistically rather than deterministically defining them. There just isn't any way for us to know how long our phrases are going to be, and how many of them we'll need.

Composobot accounts for this in a relatively simple way. Starting at the beginning of a piece and as in Chord Progression generation, Composobot will generate a first

melody rhythm pattern according to the distribution of conditional probabilities of melody rhythm patterns given an abstract start symbol. Melody rhythm patterns will continue to be generated according to the Melody Phrase Rhythm Pattern transition matrix. Each new melody rhythm pattern will be considered to start at the beginning of the next measure following the preceding pattern, since melody rhythm patterns are defined relative to the start of a measure, which is to say that the patterns were learned such that they model start-of-measure rests and so can naively be aligned with measures here, in the composition step.

As Composobot generates a progression of melody rhythm patterns, they keep track of how many measures have been covered by the progression so far. When a new pattern would be generated that begins in the kth measure of the piece, where k is greater than the number of measures covered by the chord progression, that rhythm pattern is instead discarded and the Melodic Phrase Progression is considered complete.

At this point we have the skeleton of melody in terms of rhythm. All that remains is to assign notes to that rhythmic skeleton, and we'll have a composition ready to listen to!

7.6 Melody

This is where we bring it all together. At this point, we have a chord progression that covers the entire piece, as well as a progression of melodic phrases in terms of rhythm. Now we will use both of those progressions, together with the Melody Pitch matrix from our model corresponding to the selected Mode of our piece, to build a final melody.

Recall that entries in the Melody Pitch transition matrix are calculated conditional probabilities of the form $P(Pitch \mid PrecedingPitch, UnderlyingChord)$: for example, $P(C5 \mid A4, \text{“}Am\text{”})$.

For each melody rhythm pattern in our progression, Composobot will select the chord from the chord progression that corresponds to the point in the piece where the melody rhythm pattern begins, call it X_1. Then the first pitch p_1 will be selected according to the distribution for $P(p_j \mid S, X_1)$, where S is the abstract start symbol and j is the number of observed pitches satisfying the conditions. For each subsequent position i in a melody rhythm pattern, the chord progression will again be examined

to determine the current chord X_i at that point, and the next pitch will be selected according to $P(p_j \mid p_{i-1}, X_i)$. This proceeds until the last pitch for each melody rhythm pattern is selected.

Because of the chord substitution step Composobot has performed, as described in Section 7.3, it is possible that some combination of pitch and chord we are considering was never observed in the corpus, and so is not found in the Melody Pitch matrix. In the event that there are no pitches p_j satisfying a set of conditions, which is to say that $j = 0$ for that set of conditions, then the conditions are relaxed and the next pitch is sampled from $P(p_j \mid p_{i-1})$. This has been observed to occur only very rarely, and is a compromise that both Composobot and the author are comfortable with.

Once this step is complete, we have it all: a chord progression that has been converted into a list of accompaniment notes, and a melody that has both rhythm and pitch. If we've done our job right, the melody has been composed so that it complements the accompaniment in the same way melody complemented accompaniment in our corpus. If we've done our job right, we've created music!

The very final step that Composobot performs, their victory lap if you will, is to take the list of accompaniment notes and the list of melody notes and write them to two MIDI tracks, and then write those MIDI tracks out to a MIDI file. This file will play those notes as audio, and we can finally listen to what we've been working so hard to produce: a novel musical composition!

Chapter 8

Conclusions and Discussion

At this point, a woeful tragedy must be bemoaned: Composobot's exciting final output is audio, which lends itself not well at all to enjoyment via the written page. How can we enjoy the delightful denouement of discussion without the triumphant climax of hearing the music Composobot has labored so laboriously to produce?

There are several options available to us here. The first would be to simply not offer any output; this is the worst option by far, and nobody would be more disappointed than Composobot and their author, who indulge in some pride in what Composobot is capable of producing. Bad option, let's toss it.

The second is to host some examples of output somewhere on the internet and make them available for download. This sounds great, and is certainly done: examples of input and output MIDI files can be found at http://www.d.umn.edu/~mhampton/Composobot/. However, this is unlikely to be sufficient. In the course of the research that informs this work, we discovered that many works host such files, presumably with the very best of intentions, only to find that domains expire and pages cease to exist, being more susceptible than the written page to time's inexorable flight. If you, the reader of some future age, find yourself so rebuffed, I hope there is some comfort in knowing that we predicted your disappointment and made an effort to forestall it.

Due to this unreliability, we will also undertake a third approach. In Appendix *C* you will find sample output in the form of sheet music. This format has its limitations: unless you have some musical training, it will be difficult to translate what is on the page into music; unless you have substantial musical training, it will probably be impossible

to read the output and have the experience of hearing the music directly. In the parlance we are now so accustomed to thinking in, the musical information here is durably stored, but difficult to retrieve.

The final recourse, the most powerful as well as the least accessible, is found in Appendix *A*: Composobot's complete and unabridged Julia source code. In the final extremity of desperation to experience Composobot's output for yourself, it would be possible to compile this code and have a Composobot of your very own. You could feed it all of the well-formatted MIDI files you want and experience output nobody else ever will. Whether the experience is more edifying when shared or when exclusive is outside of the scope of this work; that question is a journey whose peaks and valleys are uncharted.

8.1 Discussion

At the outset of this grand endeavor, the aim was to evaluate whether these techniques, some inspired by work that has come before and some invented entirely for this undertaking, could be combined in a way that could both learn essential patterns in how music is composed and use those learned patterns to compose novel musical pieces. The answer to this is a resounding affirmative: Composobot works, and that discovery is unspeakably edifying!

However, there is a higher bar that we hoped to leap, a grander aspiration and more interesting question: how much of the magic that is in the music we love is also in the music Composobot creates? The answer to this one is necessarily subjective, but I will offer my own subjective appraisal, as I expect you also shall. That answer is something of a relief to Composobot's author, and not unexpected: some of the magic is there, and some of it isn't. Composobot's output sounds like music, and for the most part it is pleasant to listen to, but it lacks something as well.

Much in the way that words can be strung together in a way that is aesthetically satisfying, creating a sort of poetry, even if the words are chosen solely for their phonemic properties and not to convey a message of any sort, Composobot's compositions are pleasant to listen to, but similarly convey no real message. Composobot has learned the form and structure of poetry, if you will, which is far from nothing- certainly I don't

intend to diminish Composobot's accomplishments or the significance of their results. Nevertheless, and unsurprisingly, Composobot seems to have nothing to say with the beautiful structure they have learned, no message to convey. Some of the magic, it would seem, is not there.

On the other hand, and as we discussed in the introduction, not all of music's magic is in the music itself: some large share of the magic is being performed by the listener! Humans bring their own subtext to every story, and everything they see becomes a story about their subtext. We see faces in every knot of wood, hear malice in the rumble of thunder, bask in the benevolence expressed by the lazy clouds of a summer sky. Anecdotally, nobody who has heard Composobot's output without knowing its origin has expressed that it's their new favorite music, but neither have they experienced it as purposeless noise, and some have expressed an enjoyment of the music and its *style*. I emphasize style because style, in some sense, implies intention of selection, a series of aesthetic choices that in itself conveys a message of sorts. In the words of Douglas Hofstadter[17],

> Perhaps works of art are trying to convey their style more than anything else. In that case, if you could ever plumb a style to its very bottom, you could dispense with the creations in that style.

An idea whose recursive beauty is worthy of Dr. Hofstadter: style is both the means of conveying the message and the message itself. In which case, who am I to say that Composobot has nothing to say with the style they have learned? Perhaps style is enough!

Nevertheless, Composobot has its limitations. Even if style is the message, there are ways Composobot could be improved and more of the style present in its training pieces captured. Some of those avenues are likely hidden to me, or I would have pursued more of them in Composobot's creation, but some are clear and known.

The first extension I would propose for Composobot or a program like them is a generalization of timing. Currently, Composobot assumes every piece they read is in $\frac{4}{4}$ time, which has required that every piece in their training corpus be in $\frac{4}{4}$ time. However, Temperley proposes an outline of a method for algorithmically detecting the time signature of a piece by considering the relative conditional probabilities of different

time signatures given the rhythm of the notes in the piece[8]. Implementing detection of time signature in Composobot's preprocessing step would be relatively straightforward, and my belief is that implementing variable time signature in the learning and composition steps would involve interesting generalization problems that, ultimately, would not break anything. This would be the very next improvement I would make to Composobot.

Another opportunity is in the way Composobot models local and global positions. Local position as a recurring count and global position as a binary measure of close proximity to the end of a piece are somewhat crude measures. I believe that more gradual measures could be devised that better capture the patterns those metrics seek to measure. It would be interesting to explore what those measures might be, and how they would change the compositions Composobot produces.

Finally, I would aspire to explore the idea of $motif$, the recurrence of melodic themes or phrases within a piece. It would be interesting to consider the detection of $motif$ within a piece, and a model for the way an abstract $motif$ is used to generate variations on itself within a piece's melody. It's possible that Composobot could be extended to generate a $motif$ for a piece it is composing, and use that $motif$ to inform the selection of melody rhythms and pitches during the composition step. This is somewhat less straightforward an extension, but certainly interesting and worthy of consideration.

Composobot ties together a history of insights from music theory, a breadth of ideas about music information retrieval, and a synthesis of those insights and ideas into a program that composes novel music. Whatever their limitations, and even regardless of the personal triumphs of the individual pieces they've composed, Composobot has accomplished an important thing: demonstrating to you, and to me, and even to themself, that this thing is possible, that some significant part of the magic that is in music, some part of the magic that is in us, can be captured this way.

References

[1] Rothstein, Joseph. MIDI: A comprehensive introduction. Vol. 7. AR Editions, Inc., 1995.

[2] Longuet-Higgins, H. Christopher, and Mark J. Steedman. "On interpreting bach." Machine intelligence 6 (1971): 221-241.

[3] Krumhansl, Carol L., and Mark Schmuckler. "A key-finding algorithm based on tonal hierarchies." Cognitive Foundations of Musical Pitch (1990): 77-110.

[4] Krumhansl, Carol L., and Edward J. Kessler. "Tracing the dynamic changes in perceived tonal organization in a spatial representation of musical keys." Psychological review 89.4 (1982): 334.

[5] Temperley, David, and Elizabeth West Marvin. "Pitch-class distribution and the identification of key." Music Perception: An Interdisciplinary Journal 25.3 (2008): 193-212.

[6] Hu, Diane, and Lawrence K. Saul. "A Probabilistic Topic Model for Unsupervised Learning of Musical Key-Profiles." ISMIR. 2009.

[7] Paiement, Jean-Franois, Douglas Eck, and Samy Bengio. "A probabilistic model for chord progressions." Proceedings of the Sixth International Conference on Music Information Retrieval (ISMIR). No. EPFL-CONF-83178. 2005.

[8] Temperley, David. Music and probability. Mit Press, 2007.

[9] Temperley, David. The cognition of basic musical structures. MIT press, 2004.

[10] Nierhaus, Gerhard. Algorithmic composition: paradigms of automated music generation. Springer Science & Business Media, 2009.

[11] Van Der Merwe, Andries, and Walter Schulze. "Music generation with Markov models." IEEE MultiMedia 18.3 (2011): 78-85.

[12] Raphael, Christopher, and Josh Stoddard. "Harmonic analysis with probabilistic graphical models." (2003).

[13] Rabiner, Lawrence R. "A tutorial on hidden Markov models and selected applications in speech recognition." Proceedings of the IEEE 77.2 (1989): 257-286.

[14] Bezanson, Jeff, et al. "Julia: A fresh approach to numerical computing." SIAM review 59.1 (2017): 65-98.

[15] Hobson, Joel. MIDI.jl. Computer software. Vers. 1.6.3. Web. https://github.com/JoelHobson/MIDI.jl.

[16] Tabit. Computer software. Tabit.net. Vers. 2.03. Web.

[17] Hofstadter, Douglas R. Gödel, Escher, Bach. New York: Vintage Books, 1980.

Appendix A

Composobot Source Code

What follows is the unabridged raw Julia source code for Composobot, which is entirely the original work of the author. This code is inadequately commented to be easily understood by reading it directly, and I would recommend that expectations on that front be kept reasonable and moderate. That said, everything Composobot is and does is contained in this code, and I invite anyone to use the whole or any part of this code for any purpose whatsoever, excepting the limitation of anyone else's use. Not all ideas are magnificent, but all ideas deserve to live free.

In [ ]:
```
using MIDI
```

Types

In [ ]:
```
type KBMode
    modename::String #Name of Mode (eg Minor)
    keys::Array #Array of 12 KBKey objects
end
```

In [ ]:
```
type KBKey
    keyname::String #Name of key (eg F)
    dist::Array #1x12 vector representing probability distribution of notes for this key
end
```

In [ ]:
```
type KBChord
    tonic::String #(eg A)
    mode::String #(mM7)
    rawname::String #(eg Am, C7)
    rep::Array #Notes of given chord
end
```

In [ ]:
```
type VectorNote
    keyname::String
    vector::Array
end
```

In [ ]:
```
type inputChord
    name::String
    vector::Array
    label::String
end
```

In [ ]:
```
type inputChordList
    chords::Array{inputChord}
end
```

In [ ]:
```
type inputChordProgression
    chords::inputChordList
    startTimes::Array{Any}
    endTimes::Array{Any}
end
```

In [ ]:
```
type chordDist
    targetchord::inputChord
    distance::Float64
end
```

In [ ]:
```
type chordDistances
    name::inputChord
    distances::Array{chordDist}
end
```

In [ ]:
```
type chordDistanceList
    list::Array{chordDistances}
end
```

In [ ]:
```
type chordSubProbs
    chord::inputChord
    probabilities::Array
end
```

In [ ]:
```
type chordSubProbList
    probabilities::Array{chordSubProbs}
    chordlist::inputChordList
end
```

In [ ]:
```
type cpkbEntry
    chordprior::Array
    localmetric::Int
    last::Int
    uniqueID::String
    dist::Array
end
```

Knowledge Base

Major Mode

In [ ]:
```
KBMode_Major = KBMode("Major",Array{KBKey}(12));
KBKey_CM = KBKey("C",[0.32, 0.01, 0.03, 0.01, 0.22, 0.04, 0.01, 0.225, 0.01, 0.03, 0.07, 0.025]);
KBKey_CsM = KBKey("C#",[0.025, 0.32, 0.01, 0.03, 0.01, 0.22, 0.04, 0.01, 0.225, 0.01, 0.03, 0.07]);
KBKey_DM = KBKey("D",[0.07, 0.025, 0.32, 0.01, 0.03, 0.01, 0.22, 0.04, 0.01, 0.225, 0.01, 0.03]);
KBKey_DsM = KBKey("D#",[0.03, 0.07, 0.025, 0.32, 0.01, 0.03, 0.01, 0.22, 0.04, 0.01, 0.225, 0.01]);
KBKey_EM = KBKey("E",[0.01, 0.03, 0.07, 0.025, 0.32, 0.01, 0.03, 0.01, 0.22, 0.04, 0.01, 0.225]);
KBKey_FM = KBKey("F",[0.225, 0.01, 0.03, 0.07, 0.025, 0.32, 0.01, 0.03, 0.01, 0.22, 0.04, 0.01]);
KBKey_FsM = KBKey("F#",[0.01, 0.225, 0.01, 0.03, 0.07, 0.025, 0.32, 0.01, 0.03, 0.01, 0.22, 0.04]);
KBKey_GM = KBKey("G",[0.04, 0.01, 0.225, 0.01, 0.03, 0.07, 0.025, 0.32, 0.01, 0.03, 0.01, 0.22]);
KBKey_GsM = KBKey("G#",[0.22, 0.04, 0.01, 0.225, 0.01, 0.03, 0.07, 0.025, 0.32, 0.01, 0.03, 0.01]);
KBKey_AM = KBKey("A",[0.01, 0.22, 0.04, 0.01, 0.225, 0.01, 0.03, 0.07, 0.025, 0.32, 0.01, 0.03]);
KBKey_AsM = KBKey("A#",[0.03, 0.01, 0.22, 0.04, 0.01, 0.225, 0.01, 0.03, 0.07, 0.025, 0.32, 0.01]);
KBKey_BM = KBKey("B",[0.01, 0.03, 0.01, 0.22, 0.04, 0.01, 0.225, 0.01, 0.03, 0.07, 0.025, 0.32]);
KBMode_Major.keys=[KBKey_CM,KBKey_CsM,KBKey_DM,KBKey_DsM,KBKey_EM,KBKey_FM,KBKey_FsM,KBKey_GM,KBKey_GsM,KBKey_AM,
KBKey_AsM,KBKey_BM];
```

Minor Mode

In [ ]:
```
KBMode_Minor = KBMode("Minor",Array{KBKey}(12));
KBKey_Cm = KBKey("C",[0.17, 0.01, 0.13, 0.17, 0.01, 0.12, 0.01, 0.16, 0.1, 0.02, 0.03, 0.07]);
KBKey_Csm = KBKey("C#",[0.07, 0.17, 0.01, 0.13, 0.17, 0.01, 0.12, 0.01, 0.16, 0.1, 0.02, 0.03]);
KBKey_Dm = KBKey("D",[0.03, 0.07, 0.17, 0.01, 0.13, 0.17, 0.01, 0.12, 0.01, 0.16,
0.1, 0.02]);
KBKey_Dsm = KBKey("D#",[0.02, 0.03, 0.07, 0.17, 0.01, 0.13, 0.17, 0.01, 0.12, 0.01,
0.16, 0.1]);
KBKey_Em = KBKey("E",[0.1, 0.02, 0.03, 0.07, 0.17, 0.01, 0.13, 0.17, 0.01, 0.12,
0.01, 0.16]);
KBKey_Fm = KBKey("F",[0.16, 0.1, 0.02, 0.03, 0.07, 0.17, 0.01, 0.13, 0.17, 0.01,
0.12, 0.01]);
KBKey_Fsm = KBKey("F#",[0.01, 0.16, 0.1, 0.02, 0.03, 0.07, 0.17, 0.01, 0.13, 0.17,
0.01, 0.12]);
KBKey_Gm = KBKey("G",[0.12, 0.01, 0.16, 0.1, 0.02, 0.03, 0.07, 0.17, 0.01, 0.13,
0.17, 0.01]);
KBKey_Gsm = KBKey("G#",[0.01, 0.12, 0.01, 0.16, 0.1, 0.02, 0.03, 0.07, 0.17, 0.01,
0.13, 0.17]);
KBKey_Am = KBKey("A",[0.17, 0.01, 0.12, 0.01, 0.16, 0.1, 0.02, 0.03, 0.07, 0.17,
0.01, 0.13]);
KBKey_Asm = KBKey("A#",[0.13, 0.17, 0.01, 0.12, 0.01, 0.16, 0.1, 0.02, 0.03, 0.07,
0.17, 0.01]);
KBKey_Bm = KBKey("B",[0.01, 0.13, 0.17, 0.01, 0.12, 0.01, 0.16, 0.1, 0.02, 0.03,
0.07, 0.17]);
KBMode_Minor.keys=[KBKey_Cm,KBKey_Csm,KBKey_Dm,KBKey_Dsm,KBKey_Em,KBKey_Fm,KBKey_Fsm,KBKey_Gm,KBKey_Gsm,KBKey_Am,
KBKey_Asm,KBKey_Bm];
```

Major Chords

In [ ]:
```
KBChord_CM = KBChord("C","M","CM",[1,0,0,0,1,0,0,1,0,0,0,0]);
KBChord_CsM = KBChord("C#","M","C#M",[0,1,0,0,0,1,0,0,1,0,0,0]);
KBChord_DM = KBChord("D","M","DM",[0,0,1,0,0,0,1,0,0,1,0,0]);
KBChord_DsM = KBChord("D#","M","D#M",[0,0,0,1,0,0,0,1,0,0,1,0]);
KBChord_EM = KBChord("E","M","EM",[0,0,0,0,1,0,0,0,1,0,0,1]);
KBChord_FM = KBChord("F","M","FM",[1,0,0,0,0,1,0,0,0,1,0,0]);
KBChord_FsM = KBChord("F#","M","F#M",[0,1,0,0,0,0,1,0,0,0,1,0]);
KBChord_GM = KBChord("G","M","GM",[0,0,1,0,0,0,0,1,0,0,0,1]);
KBChord_GsM = KBChord("G","M","G#M",[1,0,0,1,0,0,0,0,1,0,0,0]);
KBChord_AM = KBChord("A","M","AM",[0,1,0,0,1,0,0,0,0,1,0,0]);
KBChord_AsM = KBChord("A#","M","A#M",[0,0,1,0,0,1,0,0,0,0,1,0]);
KBChord_BM = KBChord("B","M","BM",[0,0,0,1,0,0,1,0,0,0,0,1]);
```

In [ ]:
```
xKBChord_CM = KBChord("C","M","CM",[1,0,0,0,1,0,0,1,0,0,0,1]);
xKBChord_CsM = KBChord("C#","M","C#M",[1,1,0,0,0,1,0,0,1,0,0,0]);
xKBChord_DM = KBChord("D","M","DM",[0,1,1,0,0,0,1,0,0,1,0,0]);
xKBChord_DsM = KBChord("D#","M","D#M",[0,0,1,1,0,0,0,1,0,0,1,0]);
xKBChord_EM = KBChord("E","M","EM",[0,0,0,1,1,0,0,0,1,0,0,1]);
xKBChord_FM = KBChord("F","M","FM",[1,0,0,0,1,1,0,0,0,1,0,0]);
xKBChord_FsM = KBChord("F#","M","F#M",[0,1,0,0,0,1,1,0,0,0,1,0]);
xKBChord_GM = KBChord("G","M","GM",[0,0,1,0,0,0,1,1,0,0,0,1]);
xKBChord_GsM = KBChord("G","M","G#M",[1,0,0,1,0,0,0,1,1,0,0,0]);
xKBChord_AM = KBChord("A","M","AM",[0,1,0,0,1,0,0,0,1,1,0,0]);
xKBChord_AsM = KBChord("A#","M","A#M",[0,0,1,0,0,1,0,0,0,1,1,0]);
xKBChord_BM = KBChord("B","M","BM",[0,0,0,1,0,0,1,0,0,0,1,1]);
```

Minor Chords

In [ ]:
```
KBChord_Cm = KBChord("C","m","Cm",[1,0,0,1,0,0,0,1,0,0,0,0]);
KBChord_Csm = KBChord("C#","m","C#m",[0,1,0,0,1,0,0,0,1,0,0,0]);
KBChord_Dm = KBChord("D","m","Dm",[0,0,1,0,0,1,0,0,0,1,0,0]);
KBChord_Dsm = KBChord("D#","m","D#m",[0,0,0,1,0,0,1,0,0,0,1,0]);
KBChord_Em = KBChord("E","m","Em",[0,0,0,0,1,0,0,1,0,0,0,1]);
KBChord_Fm = KBChord("F","m","Fm",[1,0,0,0,0,1,0,0,1,0,0,0]);
KBChord_Fsm = KBChord("F#","m","F#m",[0,1,0,0,0,0,1,0,0,1,0,0]);
KBChord_Gm = KBChord("G","m","Gm",[0,0,1,0,0,0,0,1,0,0,1,0]);
KBChord_Gsm = KBChord("G","m","G#m",[0,0,0,1,0,0,0,0,1,0,0,1]);
KBChord_Am = KBChord("A","m","Am",[1,0,0,0,1,0,0,0,0,1,0,0]);
KBChord_Asm = KBChord("A#","m","A#m",[0,1,0,0,0,1,0,0,0,0,1,0]);
KBChord_Bm = KBChord("B","m","Bm",[0,0,1,0,0,0,1,0,0,0,0,1]);
```

In [ ]:
```
xKBChord_Cm = KBChord("C","m","Cm",[1,0,0,1,0,0,0,1,0,0,1,0]);
xKBChord_Csm = KBChord("C#","m","C#m",[0,1,0,0,1,0,0,0,1,0,0,1]);
xKBChord_Dm = KBChord("D","m","Dm",[1,0,1,0,0,1,0,0,0,1,0,0]);
xKBChord_Dsm = KBChord("D#","m","D#m",[0,1,0,1,0,0,1,0,0,0,1,0]);
xKBChord_Em = KBChord("E","m","Em",[0,0,1,0,1,0,0,1,0,0,0,1]);
xKBChord_Fm = KBChord("F","m","Fm",[1,0,0,1,0,1,0,0,1,0,0,0]);
xKBChord_Fsm = KBChord("F#","m","F#m",[0,1,0,0,1,0,1,0,0,1,0,0]);
xKBChord_Gm = KBChord("G","m","Gm",[0,0,1,0,0,1,0,1,0,0,1,0]);
xKBChord_Gsm = KBChord("G","m","G#m",[0,0,0,1,0,0,1,0,1,0,0,1]);
xKBChord_Am = KBChord("A","m","Am",[1,0,0,0,1,0,0,1,0,1,0,0]);
xKBChord_Asm = KBChord("A#","m","A#m",[0,1,0,0,0,1,0,0,1,0,1,0]);
xKBChord_Bm = KBChord("B","m","Bm",[0,0,1,0,0,0,1,0,0,1,0,1]);
```

Dominant7 Chords

In [ ]:
```
KBChord_C7 = KBChord("C","7","C7",[1,0,0,0,1,0,0,1,0,0,1,0]);
KBChord_Cs7 = KBChord("C#","7","C#7",[0,1,0,0,0,1,0,0,1,0,0,1]);
KBChord_D7 = KBChord("D","7","D7",[1,0,1,0,0,0,1,0,0,1,0,0]);
KBChord_Ds7 = KBChord("D#","7","D#7",[0,1,0,1,0,0,0,1,0,0,1,0]);
KBChord_E7 = KBChord("E","7","E7",[0,0,1,0,1,0,0,0,1,0,0,1]);
KBChord_F7 = KBChord("F","7","F7",[1,0,0,1,0,1,0,0,0,1,0,0]);
KBChord_Fs7 = KBChord("F#","7","F#7",[0,1,0,0,1,0,1,0,0,0,1,0]);
KBChord_G7 = KBChord("G","7","G7",[0,0,1,0,0,1,0,1,0,0,0,1]);
KBChord_Gs7 = KBChord("G","7","G#7",[1,0,0,1,0,0,1,0,1,0,0,0]);
KBChord_A7 = KBChord("A","7","A7",[0,1,0,0,1,0,0,1,0,1,0,0]);
KBChord_As7 = KBChord("A#","7","A#7",[0,0,1,0,0,1,0,0,1,0,1,0]);
KBChord_B7 = KBChord("B","7","B7",[0,0,0,1,0,0,1,0,0,1,0,1]);
```

Chord Label Array

In [ ]:
```
KBLabels = [KBChord_CM, KBChord_CsM, KBChord_DM, KBChord_DsM, KBChord_EM, KBChord_FM, KBChord_FsM, KBChord_GM, KB
Chord_GsM, KBChord_AM, KBChord_AsM, KBChord_BM, KBChord_Cm, KBChord_Csm, KBChord_Dm, KBChord_Dsm, KBChord_Em, KBC
hord_Fm, KBChord_Fsm, KBChord_Gm, KBChord_Gsm, KBChord_Am, KBChord_Asm, KBChord_Bm, KBChord_C7, KBChord_Cs7, KBCh
ord_D7, KBChord_Ds7, KBChord_E7, KBChord_F7, KBChord_Fs7, KBChord_G7, KBChord_Gs7, KBChord_A7, KBChord_As7, KBCho
rd_B7];
```

In [ ]:
```
xKBLabels = [xKBChord_CM, xKBChord_CsM, xKBChord_DM, xKBChord_DsM, xKBChord_EM, xKBChord_FM, xKBChord_FsM, xKBCho
rd_GM, xKBChord_GsM, xKBChord_AM, xKBChord_AsM, xKBChord_BM, xKBChord_Cm, xKBChord_Csm, xKBChord_Dm, xKBChord_Dsm
, xKBChord_Em, xKBChord_Fm, xKBChord_Fsm, xKBChord_Gm, xKBChord_Gsm, xKBChord_Am, xKBChord_Asm, xKBChord_Bm, KBCh
ord_C7, KBChord_Cs7, KBChord_D7, KBChord_Ds7, KBChord_E7, KBChord_F7, KBChord_Fs7, KBChord_G7, KBChord_Gs7, KBCho
rd_A7, KBChord_As7, KBChord_B7];
```

Parameters

In [ ]:
```
rho = 0.99; # Constant for calculating loudness
lambda = 5; # Constant for calculating substitution probability
chordsize = 5; # maximum number of notes to consider for labeling chord
minchordsize = 3; # if the chord has fewer notes than this, we'll consider it to be repeating the preceding chord
 again
metaoctave = 5; # tells the chordlist what octave to read the 60 defined chords in
threshold = 1.2; # Maximum chord distance cutoff
w_pspr1 = 1; # Weight factor for PSPR1 in scoring melody phrase segmentation
w_pspr2 = 1; # Weight factor for PSPR2 in scoring melody phrase segmentation
w_pspr3 = 1; # Weight factor for PSPR3 in scoring melody phrase segmentation
```

Functions

Preprocessing

In [ ]:
```
function getNoteName(y)
    x = mod(y,12)+1;
    ret = "";
    if x == 1
        ret = "C="
    elseif x == 2
        ret = "C#"
    elseif x == 3
        ret = "D="
    elseif x == 4
        ret = "D#"
    elseif x == 5
        ret = "E="
    elseif x == 6
        ret = "F="
    elseif x == 7
        ret = "F#"
    elseif x == 8
        ret = "G="
    elseif x == 9
        ret = "G#"
    elseif x == 10
        ret = "A="
    elseif x == 11
        ret = "A#"
    elseif x == 12
        ret = "B="
    else
        ret = "ERROR"
    end
end
```

In [ ]:
```
function getVectorNote(letter)
    ret = zeros(12);
    if letter == "C="
        ret = [1,0,0,0,0,0,0,0,0,0,0,0]
    elseif letter == "C#"
        ret = [0,1,0,0,0,0,0,0,0,0,0,0]
    elseif letter == "D="
        ret = [0,0,1,0,0,0,0,0,0,0,0,0]
    elseif letter == "D#"
        ret = [0,0,0,1,0,0,0,0,0,0,0,0]
    elseif letter == "E="
        ret = [0,0,0,0,1,0,0,0,0,0,0,0]
    elseif letter == "F="
        ret = [0,0,0,0,0,1,0,0,0,0,0,0]
    elseif letter == "F#"
        ret = [0,0,0,0,0,0,1,0,0,0,0,0]
    elseif letter == "G="
        ret = [0,0,0,0,0,0,0,1,0,0,0,0]
    elseif letter == "G#"
        ret = [0,0,0,0,0,0,0,0,1,0,0,0]
    elseif letter == "A="
        ret = [0,0,0,0,0,0,0,0,0,1,0,0]
    elseif letter == "A#"
        ret = [0,0,0,0,0,0,0,0,0,0,1,0]
    elseif letter == "B="
        ret = [0,0,0,0,0,0,0,0,0,0,0,1]
    end
    return ret;
end
```

In [ ]:
```
function keyDistance(key1::Array, key2::Array)
    sum = 0;
    distance = 0;

    for i in 1:12
        sum += (key1[i] - key2[i])^2;
    end

    distance = sqrt(sum);

    return distance;
end
```

In [ ]:
```
function getFirstMeasure(notes::Array{MIDI.Note})
    subnotes = filter(x->x.position < 96*4, notes);
    return subnotes;
end
```

In [ ]:
```
function makePieceVector(notes::Array{MIDI.Note})
    piecevector = zeros(12);
    l = length(notes);

    for i in 1:l
        rawnote = notes[i];
        note = getVectorNote(getNoteName(rawnote.value));
        piecevector += (1/l)*note;
    end

    return piecevector;
end
```

In [ ]:
```
function determineKey(piecevector::Array, magicnumber::Int)
    mindist = 1000;
    key = "Z#Mojnr";
    index = 0;

    for i in 1:12
        candidate = KBMode_Major.keys[i];
        distance = keyDistance(piecevector, candidate.dist);
        if magicnumber == i
            distance -= .2;
        end
        if distance < mindist
            mindist = distance;
            key = string(candidate.keyname, "Major");
            index = i;
        end
    end

    for i in 1:12
        candidate = KBMode_Minor.keys[i];
        distance = keyDistance(piecevector, candidate.dist);
        if magicnumber == i+12
            distance -= .2;
        end
        if distance < mindist
            mindist = distance;
            key = string(candidate.keyname, "Minor");
            index = i+12;
        end
    end

    return [key,index];
end
```

In [ ]:
```
function analyzeKey(notes::Array{MIDI.Note})
    piecevector = makePieceVector(notes);
    m1vector = makePieceVector(getFirstMeasure(notes));

    magicnumber = determineKey(m1vector, 0)[2];

    key = determineKey(piecevector, magicnumber)[1];

    return key;
end
```

In [ ]:
```
function calculateOffset(keyname::String)
    letter = "";
    offset = 0;

    letter = keyname[1:2];

    if letter == "C="
        offset = 0;
    elseif letter == "C#"
        offset = -1;
    elseif letter == "D="
        offset = -2;
    elseif letter == "D#"
        offset = -3;
    elseif letter == "E="
        offset = -4;
    elseif letter == "F="
        offset = -5;
    elseif letter == "F#"
        offset = -6;
    elseif letter == "G="
        offset = 5;
    elseif letter == "G#"
        offset = 4;
    elseif letter == "A="
        offset = 3;
    elseif letter == "A#"
        offset = 2;
    elseif letter == "B="
        offset = 1;
    end

    return offset;
end
```

In [ ]:
```
function transposePiece(notes::Array{MIDI.Note},track1::Array{MIDI.Note},track2::Array{MIDI.Note})

    keyname = analyzeKey(notes);
    offset = calculateOffset(keyname);
    retnotes = Array{MIDI.Note}(length(notes));
    rett1 = Array{MIDI.Note}(length(track1));
    rett2 = Array{MIDI.Note}(length(track2));

    for i in 1:length(notes)
        note = notes[i];
        newval = note.value + offset;
        note.value = newval;
        retnotes[i] = note;
    end

    for i in 1:length(track1)
        note = track1[i];
        newval = note.value + offset;
        note.value = newval;
        rett1[i] = note;
    end

    for i in 1:length(track2)
        note = track2[i];
        newval = note.value + offset;
        note.value = newval;
        rett2[i] = note;
    end

    return (retnotes,rett1,rett2);
end
```

In [ ]:
```
function processSong(filename::String)
    MIDIfile = readMIDIfile(filename);
    MIDIfile.tpq = 96;
    notes = getnotes(MIDIfile.tracks[2]).notes;
    if length(MIDIfile.tracks) > 2
        for i in 3:length(MIDIfile.tracks)
            append!(notes, getnotes(MIDIfile.tracks[i]).notes)
        end
    end

    track1 = getnotes(MIDIfile.tracks[2]).notes;
    track2 = getnotes(MIDIfile.tracks[3]).notes;

    notes = transposePiece(notes,track1,track2);
    key = analyzeKey(notes[1]);
    mode = key[3:7];

    outfile = MIDI.MIDIFile();
    track1 = MIDI.MIDITrack();
    track2 = MIDI.MIDITrack();
    MIDI.addnotes!(track1,notes[2]);
    MIDI.addnotes!(track2,notes[3]);
    push!(outfile.tracks, track1);
    push!(outfile.tracks, track2);
    outname = mode*"_"*filename[1:length(filename)-4];
    MIDI.writeMIDIfile(outname, outfile);
end
```

In [ ]:
```
function processAll(filenames::Array{String})
    len = length(filenames);
    for i in 1:len
        processSong(filenames[i]);
    end
end
```

Chords

In [ ]:
```
function chordDistance(chord1::inputChord, chord2::inputChord)
    v1 = chord1.vector;
    v2 = chord2.vector;

    intermediateSum = 0;

    for i in 1:12
        intermediateSum += (v1[i] - v2[i])^2
    end

    distance = sqrt(intermediateSum)

    return distance
end
```

In [ ]:
```
function labelChord(vchord::Array)
    label = "";
    mindist = 10000;

    for i in 1:length(KBLabels)
        delta = chordDistance(inputChord("",vchord,""), inputChord("",KBLabels[i].rep,""));
        if delta < mindist
            mindist = delta;
            label = KBLabels[i].rawname;
        end
    end

    return label;
end
```

In [ ]:
```
function calculateLoudness(chord::Array{MIDI.Note})
    loudness = zeros(length(chord));
    for i in 1:length(chord)
        n = chord[i];
        loudness[i] = rho^(div(n.value,12) + mod(n.value,12)+1);
    end
    return loudness;
end
```

In [ ]:
```
function vectorizeChord(chord::Array{MIDI.Note})
    returnVector = zeros(12);
    loudnessVector = calculateLoudness(chord);
    for i in 1:12
        for j in 1:length(chord)
            n = chord[j];
            if mod(n.value,12)+1 == i
                returnVector[i] += loudnessVector[j];
            end
        end
    end
    return returnVector;
end
```

In [ ]:
```
function calculateReductionWeight(chord::Array{MIDI.Note})
    weights = zeros(length(chord));

    label = labelChord(vectorizeChord(chord));

    for i in 1:length(chord)
        n = chord[i];
        dur = div(n.duration,96);
        weights[i] = (rho^(div(n.value,12) + mod(n.value,12)+1))*dur;
    end
    return weights;
end
```

In [ ]:
```
function reduceChord(chord::Array{MIDI.Note}, label::String)

    labelchord = filter(x -> x.rawname == label, KBLabels)[1];
    newchord = filter(x -> labelchord.rep[mod(x.value,12)+1] == 1, chord);
    returnchord = Array{MIDI.Note};
    quit = 0;
    i = 1;

    while quit == 0
        if length(newchord) >= minchordsize
            returnchord = newchord;
            quit = 1;
        else
            if labelchord.rep[i] == 1
                if !(in(MIDI.Note(60+i-1,96,0,0), newchord))
                    push!(newchord, MIDI.Note(60+i-1,96,0,0));
                end
            end
        end

        i = i+1;

        if i > 12
            i = i-12;
        end
    end

    return returnchord;
end
```

In [ ]:
```
function chordCombine(entry::inputChord, index, chordlist::inputChordList)
    oldChord = chordlist.chords[index];
    oldVector = oldChord.vector;
    newVector = entry.vector;

    newChord = inputChord(oldChord.name, zeros(12), oldChord.label)

    for i in 1:12
        newChord.vector[i] = (oldVector[i] + newVector[i])/2
    end

    return newChord;
end
```

In [ ]:
```
function getChordNames(chordlist::inputChordList)
    namelist = [];
    for i in 1:length(chordlist.chords)
        push!(namelist,(chordlist.chords[i]).name);
    end
    return namelist;
end
```

In [ ]:
```
function makeChordName(chord::Array{MIDI.Note})
    notes = Array{String}(0);
    name = "";

    if length(chord) > 0

        for i in 1:length(chord)
            n = chord[i];
            letter = getNoteName(n.value);
            octave = dec(div(n.value,12));
            notename = letter * octave;
            if length(notes) == 0
               push!(notes,notename);
            elseif !(notename in notes)
               push!(notes,notename);
            end
        end

        sort!(notes,by=last);

        for i in 1:length(notes)
            name = name * notes[i];
        end

    end

    return name
end
```

In [ ]:
```
function kbChordToInputChord(kbchord::KBChord, octave::Int)
    vector = kbchord.rep;
    label = kbchord.rawname;
    chord = Array{MIDI.Note}(0);

    for i in 1:12
        if vector[i] == 1
            value = octave*12 + (i-1);
            note = MIDI.Note(value, 96, 0, 0);
            push!(chord,note);
        end
    end

    name = makeChordName(chord);
    v = vectorizeChord(chord);

    returnchord = inputChord(name,v,label);

    return returnchord;
end
```

In [ ]:
```
function readChord(chord::Array{MIDI.Note}, chordlist::inputChordList, progression::inputChordProgression, startT
ime::Int, endTime::Int)

    v = vectorizeChord(chord);
    label = labelChord(v);

    rchord = reduceChord(chord, label);

    if length(rchord) >= minchordsize

        name = makeChordName(rchord);

        entry = inputChord(name,v,label);

    else

        entry = chordlist.chords[length(chordlist.chords)];

        name = entry.name;

    end

    push!(progression.chords.chords, entry);
    push!(progression.startTimes, startTime);
    push!(progression.endTimes, endTime);

    if length(chordlist.chords) == 0
        push!(chordlist.chords,entry)
    else
        namelist = getChordNames(chordlist);

        index = findfirst(namelist.==name);

        if index == 0
            push!(chordlist.chords,entry)
        else
            chordlist.chords[index] = chordCombine(entry,index,chordlist);
        end
    end
end
```

In [ ]:
```
function findLastPosition(notes::Array)
    positionList = []
    for i in 1:length(notes)
        push!(positionList,notes[i].position)
    end

    return maximum(positionList);
end
```

In [ ]:
```
function addKBChords(chordlist::inputChordList)
    for i in 1:length(KBLabels)
        newchord = kbChordToInputChord(KBLabels[i], metaoctave);
        push!(chordlist.chords, newchord);
    end

    return chordlist;
end
```

In [ ]:
```
function processChords(notes::Vector{Note}, division::Int)
    chordlist = inputChordList([]);
    progression = inputChordProgression(inputChordList([]), [], []);
    lastPosition = findLastPosition(notes);
    i = 0;

    while i <= lastPosition
        chord = filter(x->x.position >= i && x.position < i + division, notes);
        readChord(chord, chordlist, progression, i, i+division);
        i += division;
    end

    chordlist = addKBChords(chordlist);

    retarray=[chordlist, progression];

    return retarray;
end
```

In [ ]:
```
function calculateChordDistances(chordlist::inputChordList)
    distancelist = chordDistanceList([]);

    for i in 1:length(chordlist.chords)
        chord = chordlist.chords[i];
        distances = chordDistances(chord,[]);
        dlist = [];

        for j in 1:length(chordlist.chords)
            dist = chordDist(chordlist.chords[j], chordDistance(chord, chordlist.chords[j]));
            push!(dlist,dist);
        end

        distances.distances = dlist;
        push!(distancelist.list, distances);
    end

    return distancelist;
end
```

In [ ]:
```
function calculateTheta(chord1::inputChord, chord2::inputChord)
    delta = chordDistance(chord1, chord2);
    theta = exp(-1*lambda*delta);
    return theta;
end
```

In [ ]:
```
function thetaSum(chord::inputChord, chordlist::inputChordList)
    sum = 0;
    for i in 1:length(chordlist.chords)
        sum += calculateTheta(chord, chordlist.chords[i]);
    end
    return sum;
end
```

In [ ]:
```
function probSubstitution(chord1::inputChord, chord2::inputChord, chordlist::inputChordList)
    delta = chordDistance(chord1, chord2);
    if delta < threshold
        theta = calculateTheta(chord1, chord2);
        sum = thetaSum(chord1, chordlist);
        prob = theta/sum;
        return prob;
    else
        return -1;
    end
end
```

In [ ]:
```
function chordSubProbVector(chord::inputChord, chordlist::inputChordList)
    l = length(chordlist.chords);
    v = zeros(l);
    j = 0;
    for i in 1:l
        v[i-j] = probSubstitution(chord, chordlist.chords[i], chordlist);
        if v[i-j] < 0
            deleteat!(v,i-j);
            j = j + 1;
        end
    end
    return v;
end
```

In [ ]:
```
function generateChordSubProbList(chordlist::inputChordList)
    l = length(chordlist.chords);
    CSPL = chordSubProbList(Array{chordSubProbs}(l), chordlist);
    for i in 1:l
        entry = chordSubProbs(chordlist.chords[i], chordSubProbVector(chordlist.chords[i], chordlist));
        CSPL.probabilities[i] = entry;
    end
    return CSPL;
end
```

In [ ]:
```
function cpkbFreqToProb(cpkb::Array)
    length1 = length(cpkb);

    for i in 1:length1
        dist = cpkb[i].dist;

        length2 = length(dist);
        freqsum = 0;

        for j in 1:length2
            freqsum += dist[j][2];
        end

        for j in 1:length2
            dist[j][2] = (dist[j][2])/freqsum;
        end

    end

    return cpkb
end
```

In [ ]:
```
function arraykbFreqToProb(cpkb::Array)
    length1 = length(cpkb);

    for i in 1:length1
        dist = cpkb[i][2];

        length2 = length(dist);
        freqsum = 0;

        for j in 1:length2
            freqsum += dist[j][2];
        end

        for j in 1:length2
            dist[j][2] = (dist[j][2])/freqsum;
        end

    end

    return cpkb
end
```

In [ ]:
```
function learnCP(cps::Array)
    cpkb = [];
    length1 = length(cps);

    for i in 1:length1
        cp = cps[i].chords.chords;
        length2 = length(cp);

        for j in 1:length2

            thischord = cp[j].name;

            localmetric = mod(j,4);
            if localmetric == 0
                localmetric = 4;
            end

            if (length2-j) < 4
                last = 1;
            else
                last = 0;
            end

            if j == 1
                chordprior = ["Start"];
                cpstring = chordprior[1];
            elseif j == 2
                chordprior = [cp[1].name];
                cpstring = chordprior[1]
            else
                chordprior = String[];
                push!(chordprior, cp[j-2].name);
                push!(chordprior, cp[j-1].name);
                cpstring = chordprior[1]*"_"*chordprior[2];
            end

            uniqueID = cpstring*"_"*string(localmetric)*"_"*string(last);

            index = findfirst(x -> x.uniqueID == uniqueID, cpkb);

            if index == 0
                dist = [[thischord, 1]];
                entry = cpkbEntry(chordprior, localmetric, last, uniqueID, dist);
                push!(cpkb,entry);
            else
                index2 = findfirst(x -> x[1] == thischord, cpkb[index].dist)

                if index2 == 0
                    push!(cpkb[index].dist, [thischord, 1]);
                else
                    cpkb[index].dist[index2][2] += 1;
                end

            end

        end

    end

    cpkb = cpkbFreqToProb(cpkb);

    return cpkb;
end
```

Accompaniment Rhythm

In [ ]:
```
function buildRhythmForm(chord::Vector{Note}, metricunit::Int, division::Int)
    rhythmform = [0,0,0,0,0,0,0,0];

    for i in 1:length(chord)
        note = chord[i];
        pos = trunc(Int,(mod(note.position, division))/metricunit + 1);
        rhythmform[pos] = 1;
    end

    indices = [];

    for i in 1:8
        if rhythmform[i] == 1
            push!(indices,i);
        end
    end

    push!(indices,9);

    if length(indices) > 1
        for i in 1:(length(indices) - 1)
            a = indices[i];
            diff = indices[i+1] - indices[i];
            rhythmform[a] = diff;
        end
    end

    return rhythmform;
end
```

In [ ]:
```
function learnRhythmForms(notes::Vector{Note}, division::Int)
    rhythmforms = [];
    metricunit = 48;
    lastPosition = findLastPosition(notes);
    i = 0;

    while i <= lastPosition
        chord = filter(x->x.position >= i && x.position < i + division, notes);
        rhythmform = buildRhythmForm(chord,metricunit,division);
        push!(rhythmforms,rhythmform);
        i += division;
    end

    return rhythmforms;
end
```

In [ ]:
```
function rhythmFormToString(rhythmform::Array)
    retstring = "";

    for i in 1:8
        retstring = retstring*string(rhythmform[i]);
    end

    return retstring;
end
```

In [ ]:
```
function learnRhythmCP(rhythmform_proglist::Array)
    rpkb = [];
    length1 = length(rhythmform_proglist);

    for i in 1:length1
        rp = rhythmform_proglist[i];
        length2 = length(rp);

        for j in 1:length2

            rhythmform = rp[j];

            localmetric = mod(j,4);
            if localmetric == 0
                localmetric = 4;
            end

            if (length2-j) < 4
                last = 1;
            else
                last = 0;
            end

            if j == 1
                rhythmform_prior = ["Start"];
                rpstring = rhythmform_prior[1];
            else
                rhythmform_prior = rp[j-1];
                rpstring = rhythmFormToString(rhythmform_prior);
            end

            uniqueID = rpstring*"_"*string(localmetric)*"_"*string(last);

            index = findfirst(x -> x.uniqueID == uniqueID, rpkb);

            if index == 0
                dist = [[rhythmform, 1]];
                entry = cpkbEntry(rhythmform_prior, localmetric, last, uniqueID, dist);
                push!(rpkb,entry);
            else
                index2 = findfirst(x -> x[1] == rhythmform, rpkb[index].dist)

                if index2 == 0
                    push!(rpkb[index].dist, [rhythmform, 1]);
                else
                    rpkb[index].dist[index2][2] += 1;
                end

            end

        end

    end

    rpkb = cpkbFreqToProb(rpkb);

    return rpkb;
end
```

Learning and Generation

Assumption: preprocessing gives us files transposed to C, with a time division of 96 ticks, with all melody notes in track 1 and all accompaniment notes in track 2.

In [ ]:
```
function readAllChords(filenames::Array{String})
    l = length(filenames);
    chordlist = [];
    proglist_m = [];
    proglist_M = [];
    melody_proglist_m = [];
    melody_proglist_M = [];
    rhythmform_proglist = [];

    for i in 1:l
        filename = filenames[i];
        MIDIfile = readMIDIfile(filename);
        melody = getnotes(MIDIfile.tracks[1]).notes;
        notes = getnotes(MIDIfile.tracks[1]).notes;
        append!(notes,getnotes(MIDIfile.tracks[2]).notes);
        chord_division = 96*4;
        mode = filename[1:5];

        results = processChords(notes, chord_division);
        chords = results[1].chords;
        prog = results[2];

        plength = max(melody[length(melody)].position + melody[length(melody)].duration,
                      notes[length(notes)].position + notes[length(notes)].duration);

        plength = trunc(Int, plength);

        melody_prog = makeMelodyPhraseProgression(melody, plength);

        rhythmform_prog = learnRhythmForms(notes, chord_division);
        push!(rhythmform_proglist, rhythmform_prog);

        for j in 1:length(chords)
            if chords[j] in chordlist
                ;
            else
                push!(chordlist,chords[j]);
            end
        end

        if mode == "Major"
            push!(proglist_M, prog);
            push!(melody_proglist_M, melody_prog);
        elseif mode == "Minor"
            push!(proglist_m, prog);
            push!(melody_proglist_m, melody_prog);
        else
            print("YOU HECKED UP, SOME FILE ISN'T PREPROCESSED\n");
        end
    end

    cpkb_M = learnCP(proglist_M);
    cpkb_m = learnCP(proglist_m);
    substitution_probs = generateChordSubProbList(inputChordList(chordlist));
    rhythm_probs = learnRhythmCP(rhythmform_proglist);
    melody_cpkb_M = learnPhraseCPs(melody_proglist_M, proglist_M);
    melody_cpkb_m = learnPhraseCPs(melody_proglist_m, proglist_m);

    retarray = [cpkb_M, cpkb_m, substitution_probs, rhythm_probs, melody_cpkb_M, melody_cpkb_m];

    return retarray;
end
```

In [ ]:
```
function markovNextChord(cpkb::Array, prior::Array, localmetric::Int, last::Int)
    index = findfirst(x -> (x.chordprior == prior && x.localmetric == localmetric && x.last == last), cpkb)

    if index == 0
        index = findfirst(x -> (x.chordprior == prior && x.localmetric == localmetric), cpkb);
    end

    if index == 0
        index = findfirst(x -> (x.chordprior == prior), cpkb);
    end

    if index == 0
        index = findfirst(x -> (x.localmetric == localmetric), cpkb);
    end

    entry = cpkb[index];
    dist = entry.dist;

    n = rand();
    runningsum = 0;
    next = "";

    for i in 1:length(dist)
        runningsum += dist[i][2];
        if n < runningsum
            next = dist[i][1];
            break;
        end
    end

    return(next);
end
```

Assumption: plength is greater than or equal to 8

In [ ]:
```
function markovSelectChords(cpkb::Array, plength::Int)
    prog = Array{String}(plength);
    last = 0;

    prog[1] = markovNextChord(cpkb, ["Start"], 1, last);
    prog[2] = markovNextChord(cpkb, [prog[1]], 2, last);

    for i in 3:plength

        if (plength - i) < 4
            last = 1;
        end

        localmetric = mod(i,4);
        if localmetric == 0
            localmetric = 4;
        end

        priors = [prog[i-2], prog[i-1]];

        prog[i] = markovNextChord(cpkb, priors, localmetric, last);

    end

    return prog;
end
```

In [ ]:
```
function markovSelectRhythms(rpkb::Array, plength::Int)
    prog = Array{Array}(plength);
    last = 0;

    prog[1] = markovNextChord(rpkb, ["Start"], 1, last);
    prog[2] = markovNextChord(rpkb, [prog[1]], 2, last);

    for i in 3:plength

        if (plength - i) < 4
            last = 1;
        end

        localmetric = mod(i,4);
        if localmetric == 0
            localmetric = 4;
        end

        priors = [prog[i-2], prog[i-1]];

        prog[i] = markovNextChord(rpkb, priors, localmetric, last);

    end

    return prog;
end
```

In [ ]:
```
function pickSubChord(dist::Array)
    n = rand();
    runningsum = 0;
    index = 1;

    for i in 1:length(dist)
        runningsum += dist[i];
        if n < runningsum
            index = i;
            break;
        end
    end

    return index;
end
```

In [ ]:
```
function substituteChords(prog::Array, subprobs::chordSubProbList)
    outprog = Array{String}(length(prog));
    probs = subprobs.probabilities;

    chord = "";

    for i in 1:length(prog)
        name = prog[i];
        index = findfirst(x -> x.chord.name == name, probs);
        if index == 0
            index = 1;
        end
        entry = probs[index];
        dist = entry.probabilities;

        index = pickSubChord(dist);
        chord = probs[index].chord.name;
        outprog[i] = chord;
    end

    return outprog;
end
```

In [ ]:
```
function generateProgression(cpkb_M::Array, cpkb_m::Array, substitution_probs::chordSubProbList, mode::String, pl
ength::Int)

    if mode == "Major"
        cpkb = cpkb_M;
    elseif mode == "Minor"
        cpkb = cpkb_m;
    else
        print("Error! Bad mode argument!\n");
    end

    prog = markovSelectChords(cpkb, plength);
    prog = substituteChords(prog, substitution_probs);

    return prog;
end
```

In [ ]:
```
function generateRhythmProg(rpkb::Array, plength::Int)
    rhythm_prog = markovSelectRhythms(rpkb, plength);

    return rhythm_prog;
end
```

In [ ]:
```
function getNoteValue(name::String)
    value = 2000;

    if name == "C="
        value = 0;
    elseif name == "C#"
        value = 1;
    elseif name == "D="
        value = 2;
    elseif name == "D#"
        value = 3;
    elseif name == "E="
        value = 4;
    elseif name == "F="
        value = 5;
    elseif name == "F#"
        value = 6;
    elseif name == "G="
        value = 7;
    elseif name == "G#"
        value = 8;
    elseif name == "A="
        value = 9;
    elseif name == "A#"
        value = 10;
    elseif name == "B="
        value = 11;
    else
        print("Error in getNoteValue\n");
    end

    return value;
end
```

In [ ]:
```
function generateWithRhythm(notelist::Array, rhythmform::Array, ordinality::Int)
    returnnotes = [];

    for i in 1:8
        n = rand();
        if rhythmform[i] != 0
            dur = (96*rhythmform[i])/2
            pos = (ordinality-1)*96*4 + (96*i)/2
            if mod(i,8) == 1
                push!(returnnotes, MIDI.Note(notelist[1], dur, pos, 0))
                if n > 0.5
                    push!(returnnotes, MIDI.Note(notelist[2], dur, pos, 0))
                end
            elseif mod(i,4) == 1
                push!(returnnotes, MIDI.Note(notelist[1], dur, pos, 0))
                push!(returnnotes, MIDI.Note(notelist[2], dur, pos, 0))
                if (n > 0.5 && length(notelist) > 2)
                    push!(returnnotes, MIDI.Note(notelist[3], dur, pos, 0))
                end
            elseif mod(i,2) == 1
                for j in 3:length(notelist)
                    push!(returnnotes, MIDI.Note(notelist[j], dur, pos, 0))
                end
            else
                if n < 0.33
                    push!(returnnotes, MIDI.Note(notelist[1], dur, pos, 0))
                elseif n < 0.66
                    push!(returnnotes, MIDI.Note(notelist[2], dur, pos, 0))
                else
                    push!(returnnotes, MIDI.Note(notelist[length(notelist)], dur, pos, 0))
                end
            end
        end
    end

    return returnnotes;
end
```

In [ ]:
```
function chordNameToNotes(name::String, rhythmform::Array, offset::Int, ordinality::Int)
    l = length(name)/3;
    divlength = 96*2;

    notelist = [];
    returnnotes = [];

    for i in 1:l
        index = Int(3*(i-1) + 1);
        notename = name[index:(index+2)];

        note = getNoteValue(notename[1:2]);
        oct = parse(Int, notename[3]);

        note = (note + oct*12) + offset;

        push!(notelist,note)

    end

    returnnotes = generateWithRhythm(notelist, rhythmform, ordinality);

    return returnnotes;
end
```

In [ ]:
```
function makeChordLabel(name::String)
    l = trunc(Int,length(name)/3);

    notelist = Array{MIDI.Note}(l);

    for i in 1:l
        index = Int(3*(i-1) + 1);
        notename = name[index:(index+2)];

        note = getNoteValue(notename[1:2]);
        oct = parse(Int, notename[3]);

        note = MIDI.Note((note + oct*12) + offset,96,0,0);

        #push!(notelist, note) ;
        notelist[i] = note;
    end

    label = labelChord(vectorizeChord(notelist));

    return(label);
end
```

In [ ]:
```
function outputProgression(prog::Array, rhythms::Array, melody::Array, offset::Int, outname::String)

    outfile = MIDI.MIDIFile();
    track1 = MIDI.MIDITrack();
    notes1 = MIDI.Note[];
    notes2 = MIDI.Note[];

    l = length(prog);

    for i in 1:length(melody)
        note = melody[i];
        note.value = note.value + offset;
        push!(notes1, note);
    end

    for i in 1:l
        chordnotes = chordNameToNotes(prog[i], rhythms[i], offset, i);
        for j in 1:length(chordnotes)
            push!(notes2,chordnotes[j]);
        end
    end

    last = notes2[length(notes2)].position + notes2[length(notes2)].duration;
    buffernote = MIDI.Note(0, 96*2, last, 0, 0);
    push!(notes2,buffernote);

    MIDI.addnotes!(track1, notes1);
    MIDI.addnotes!(track1, notes2);
    push!(outfile.tracks, track1);
    MIDI.writeMIDIfile(outname, outfile);
    print(outname*" written and ready to be listened to!\n");

end
```

Melody

In [ ]:
```
function IOI(note1::Note, note2::Note)
    return (note2.position - note1.position);
end
```

In [ ]:
```
function OOI(note1::Note, note2::Note)
    return (note2.position - (note1.position + note1.duration));
end
```

In [ ]:
```
function getmeanIOI(notes::Vector{Note})
    IOIs = [];

    for i in 1:(length(notes)-1)
        push!(IOIs, IOI(notes[i+1],notes[i]));
    end

    meanIOI = mean(IOIs);

    return trunc(Int,meanIOI);
end
```

In [ ]:
```
function getPSPR1(note1::Note, note2::Note, meanIOI::Int)
    return w_pspr1*((IOI(note2,note1) + OOI(note2,note1))/meanIOI);
end
```

In [ ]:
```
function getPSPR2(n::Int)
    return w_pspr2*(-abs(log2(n) - 3));
end
```

In [ ]:
```
function getPSPR3(note::Note)
    return w_pspr3*(-log2((mod(note.position,96*4)*16/(96*4))+0.99));
end
```

In [ ]:
```
function getPSPRscore(phrase1::Vector{Note}, phrase2::Vector{Note}, meanIOI::Int)
    PSPR1 = getPSPR1(phrase1[length(phrase1)], phrase2[1], meanIOI);
    PSPR2 = getPSPR2(phrase2);
    PSPR3 = getPSPR3(phrase2[1]);

    return PSPR1 + PSPR2 + PSPR3;
end
```

In [ ]:
```
function makePSPR1vector(notes::Vector{Note})
    PSPR1vector = [];
    meanIOI = getmeanIOI(notes);

    for i in 1:(length(notes)-1)
        note1 = notes[i];
        note2 = notes[i+1];
        push!(PSPR1vector, getPSPR1(note1, note2, meanIOI));
    end

    return PSPR1vector;
end
```

In [ ]:
```
function makePSPR3vector(notes::Vector{Note})
    PSPR3vector = [];

    push!(PSPR3vector,0);

    for i in 1:(length(notes)-1)
        note = notes[i+1];
        push!(PSPR3vector, getPSPR3(note));
    end

    return PSPR3vector;
end
```

In [ ]:
```
function findMPdivisions(scores::Array)
    MPdivisions = [];
    remainingScores = scores;

    quit = 0;
    index = 0;

    push!(MPdivisions,0);

    while quit == 0
        iteratedScores = copy(remainingScores);
        for i in 1:length(remainingScores)
            iteratedScores[i] = remainingScores[i] * getPSPR2(i);
        end

        m = trunc(Int,findmax(iteratedScores)[2]);

        push!(MPdivisions,index + m);

        index = index + m;

        if m == length(remainingScores)
            quit = 1;
        else
            remainingScores = getindex(remainingScores, (m+1):length(remainingScores));
        end
    end

    return(MPdivisions);
end
```

In [ ]:
```
function addSilentFinalNote(notes::Vector{Note})
    lastnote = notes[length(notes)];
    newpos = lastnote.position + lastnote.duration;

    diff = 96*4 - mod(newpos, 96*4);
    newpos = newpos + diff;

    newnote = MIDI.Note(0,0,newpos,0,0);
    push!(notes,newnote);
    return notes;
end
```

In [ ]:
```
function makeMelodyPhrases(notes::Vector{Note})
    melodyPhrases = [];

    notes = addSilentFinalNote(notes);

    PSPR1vector = makePSPR1vector(notes);
    PSPR3vector = makePSPR3vector(notes);

    PSPR1_3vector = [];

    for i in 1:length(PSPR1vector)
        push!(PSPR1_3vector, PSPR1vector[i] + PSPR3vector[i]);
    end

    MPdivisions = findMPdivisions(PSPR1_3vector);

    for i in 1:(length(MPdivisions)-1)
        index1 = MPdivisions[i] + 1;
        index2 = MPdivisions[i+1];
        push!(melodyPhrases, getindex(notes,index1:index2));
    end

    return melodyPhrases;
end
```

In [ ]:
```
function constructPhraseProgression(phrases::Array, plength::Int)
    phraseProgression = [];

    firstOOI = phrases[1][1].position;

    if firstOOI > 0
        restphrase = [MIDI.Note(0,firstOOI,0,0,0)];
        push!(phraseProgression, restphrase);
    end

    push!(phraseProgression, phrases[1]);

    for i in 2:length(phrases)
        note1 = phrases[i-1][length(phrases[i-1])];
        note2 = phrases[i][1];
        ooi = OOI(note1, note2);
        if ooi > 0
            restphrase = [MIDI.Note(0,ooi,note1.position + note1.duration,0,0)];
            push!(phraseProgression, restphrase);
        end
        push!(phraseProgression, phrases[i]);
    end

    lastnote = phrases[length(phrases)][length(phrases[length(phrases)])];
    lastend = lastnote.position + lastnote.duration;
    diff = plength - lastend;

    if diff > 0
        restphrase = [MIDI.Note(0,diff,lastend,0,0)];
        push!(phraseProgression, restphrase);
    end

    return phraseProgression;
end
```

In [ ]:
```
function makeMelodyPhraseProgression(notes::Vector{Note}, plength::Int)
    melodyphrases = makeMelodyPhrases(notes);

    phraseProgression = constructPhraseProgression(melodyphrases, plength);

    return phraseProgression;
end
```

In [ ]:
```
function makePhraseRhythmID(phrase::Array)
    id = "";
    firstpos = phrase[1].position;

    for i in 1:(length(phrase))
        note = phrase[i];
        dur = string(note.duration);
        pos = string(note.position - firstpos);
        id = id*dur*";"*pos;
        if i < length(phrase)
            id = id*"|";
        end
    end

    return(id);
end
```

In [ ]:
```
function learnPhraseRhythmCP(phrase_proglist::Array)
    prkb = [];
    length1 = length(phrase_proglist);

    for i in 1:length1
        pr = phrase_proglist[i];
        length2 = length(pr);

        for j in 1:length2

            phrase = pr[j];
            rest = 0;

            if phrase[1].velocity == 0
                rest = 1;
            end

            if rest == 0
                label = makePhraseRhythmID(phrase);
                prhythm = [];
                firstpos = phrase[1].position;

                for k in 1:(length(phrase))
                    note = phrase[k];
                    prentry = [note.duration, note.position - firstpos];
                    push!(prhythm, prentry);
                end
            else
                prhythm = ["REST"];
            end

            if j == 1
                prior = "Start";
            else
                if pr[j-1][1].velocity == 0
                    prior = "REST";
                else
                    prior = makePhraseRhythmID(pr[j-1]);
                end
            end

            index = findfirst(x -> x[1] == prior, prkb);

            if index == 0
                dist = [[prhythm, 1]];
                entry = [prior,dist];
                push!(prkb,entry);
            else
                index2 = findfirst(x -> x[1] == prhythm, prkb[index][2])

                if index2 == 0
                    push!(prkb[index][2], [prhythm, 1]);
                else
                    prkb[index][2][index2][2] += 1;
                end

            end

        end

    end

    prkb = arraykbFreqToProb(prkb);

    return prkb;
end
```

In [ ]:
```
function makeValueID(value::Int, chordname::String)
    id = string(value)*"_"*chordname;

    return id;
end
```

In [ ]:
```
function learnPhraseValueCP(phrase_proglist::Array, chord_proglist::Array)
    pvkb = [];
    length1 = length(phrase_proglist);

    chordlength = 96*4;

    for i in 1:length1
        pv = phrase_proglist[i];
        length2 = length(pv);

        prog = chord_proglist[i];

        timing = 0;
        curchord = 0;
        chordname = "NOPE";

        for j in 1:length2

            phrase = pv[j];
            rest = 0;

            if phrase[1].velocity == 0
                rest = 1;
                timing = timing + phrase[1].duration;
            end

            if rest == 0

                for k in 1:length(phrase)
                    note = phrase[k];
                    value = note.value;

                    curchord = min(length(prog.chords.chords), div(timing, chordlength) + 1);

                    timing = timing + note.duration;

                    chordname = prog.chords.chords[curchord].label;

                    if k == 1
                        prior = "Start_"*chordname;
                    else
                        prior = makeValueID(trunc(Int,phrase[k-1].value), chordname);
                    end

                    index = findfirst(x -> x[1] == prior, pvkb);

                    if index == 0
                        dist = [[value, 1.0]];
                        entry = [prior,dist];
                        push!(pvkb,entry);
                    else
                        index2 = findfirst(x -> x[1] == value, pvkb[index][2])

                        if index2 == 0
                            push!(pvkb[index][2], [value, 1.0]);
                        else
                            pvkb[index][2][index2][2] += 1;
                        end
                    end
                end

            end

        end

    end

    pvkb = arraykbFreqToProb(pvkb);

    return pvkb;
end
```

In [ ]:
```
function learnPhraseCPs(phrase_proglist::Array, chord_proglist::Array)
    phraseCPs = [];

    push!(phraseCPs, learnPhraseRhythmCP(phrase_proglist));
    push!(phraseCPs, learnPhraseValueCP(phrase_proglist, chord_proglist::Array));

    return phraseCPs;
end
```

In [ ]:
```
function generateMelody(cpkb_M::Array, cpkb_m::Array, mode::String, prog::Array)
    if mode == "Major"
        cpkb = cpkb_M;
    elseif mode == "Minor"
        cpkb = cpkb_m;
    else
        print("Error! Bad mode argument!\n");
    end

    rhythmCP = cpkb[1];
    valueCP = cpkb[2];

    melody = markovMakeMelody(rhythmCP, valueCP, prog);

    return melody;
end
```

In [ ]:
```
function markovMakeMelody(rhythmCP::Array, valueCP::Array, prog::Array)

    chordlength = 96*4;

    plength = chordlength*length(prog);

    dpf_list = markovMakeDPFlist(rhythmCP, plength);

    values = markovMakeMelValues(valueCP, dpf_list, prog);

    melody = constructMelody(dpf_list, values);

    return melody;
end
```

In [ ]:

```
function markovMakeDPFlist(rhythmCP::Array, plength::Int)
    dpf_list = [];

    phrase_list = [];

    quit = 0;
    current_time = 0;
    i = 2;

    push!(phrase_list, markovNextMRPhrase(rhythmCP, ["Start"], current_time, plength));

    current_time = current_time + phrase_list[1][2];

    if current_time >= plength
        quit = 1;
    end

    while quit == 0
        prior = phrase_list[i-1][1];
        phrase = markovNextMRPhrase(rhythmCP, [prior], current_time, plength);
        push!(phrase_list, phrase);
        current_time = current_time + phrase_list[i][2];

        if current_time >= plength
            quit = 1;
        end

        i += 1;
    end

    running_position = 0;

    for j in 1:length(phrase_list)
        phrase = phrase_list[j][1];
        l = phrase_list[j][2];

        if (phrase[1] != "REST" && phrase != "REST")

            for k in 1:length(phrase)

                d = phrase[k][1];
                p = phrase[k][2] + running_position;
                f = 0;
                if k == 1
                    f = 1;
                end
                dpf = [d,p,f];

                push!(dpf_list, dpf);
            end
        end

        running_position += l;

    end

    return dpf_list;
end
```

In [ ]:
```
function markovNextMRPhrase(rhythmCP::Array, prior::Array, current_time::Int, plength::Int)
    index = findfirst(x -> (x[1] == prior[1]), rhythmCP);

    if index == 0
        index = rand(1:length(rhythmCP));
    end

    entry = rhythmCP[index];
    dist = entry[2];

    n = rand();
    runningsum = 0;
    next = "";

    for i in 1:length(dist)
        runningsum += dist[i][2];
        if n < runningsum
            next = dist[i][1];
            break;
        end
    end

    if (next == "REST" || next[1] == "REST")
        l = mod(current_time,96*4);
    else
        l = trunc(Int,next[length(next)][1] + next[length(next)][2]);
        if l > (plength - current_time)
            next = "REST";
            l = (plength - current_time);
        end
    end

    return([next,l]);
end
```

In [ ]:
```
function getPhraseLength(phrase::String)
    length = 0;

    if phrase != "REST"
        phrase_vector = split(phrase,"|");
        for i in 1:length(phrase_vector)
            dp_strings = split(phrase_vector[k],";");
            d = parse(Int,dp_strings[1]);
            length += d;
        end
    end

    return length;
end
```

In [ ]:
```
function markovMakeMelValues(valueCP::Array, dpf_list::Array, prog::Array)
    values = [];

    for i in 1:length(dpf_list)
        dpf = dpf_list[i];

        timing = dpf[1] + dpf[2];
        chord_n = div(timing, 96*4) + 1;
        chordname = makeChordLabel(prog[chord_n]);

        if dpf[3] == 1
            prior = "Start_"*chordname;
        else
            if i == 1
                print("ERROR in markovMakeMelValues: when i is 1 dpf[3] != 1\n");
            end
            prior = string(trunc(Int,values[i-1]))*"_"*chordname;
        end

        value = markovNextValue(valueCP, prior);

        push!(values,value);
    end

    return(values);
end
```

In [ ]:
```
function markovNextValue(valueCP::Array, prior::String)
    index = findfirst(x -> (x[1] == prior), valueCP);

    if index == 0
        index = findfirst(x -> (x[1][1:2] == prior[1:2]), valueCP);
    end

    entry = valueCP[index];
    dist = entry[2];

    n = rand();
    runningsum = 0;
    next = 0;

    for i in 1:length(dist)
        runningsum += dist[i][2];
        if n < runningsum
            next = dist[i][1];
            break;
        end
    end

    return(next);
end
```

In [ ]:
```
function constructMelody(dpf_list::Array, values::Array)
    melody = [];

    l = length(dpf_list);

    for i in 1:l
        note = MIDI.Note(values[i], dpf_list[i][1], dpf_list[i][2], 0);
        push!(melody,note);
    end

    return(melody);
end
```

Higher Order Writing/Reading

In [ ]:
```
function writeModel(filename::String,results::Array)
    cpkb_M = results[1];
    cpkb_m = results[2];
    sub_probs = results[3];
    rpkb = results[4];
    mel_M = results[5];
    mel_m = results[6];

    open(filename,"w") do f
        write(f, string(cpkb_M));
        write(f, "\n");

        write(f, string(cpkb_m));
        write(f, "\n");

        write(f, string(sub_probs));
        write(f, "\n");

        write(f, string(rpkb));
        write(f, "\n");

        write(f, string(mel_M));
        write(f, "\n");

        write(f, string(mel_m));
        write(f, "\n");
    end
end
```

In [ ]:
```
function readModel(filename::String)
    results = [];

    open(filename) do f
        lines = readlines(f);

        major = lines[1];
        minor = lines[2];
        subprobs = lines[3];
        rpkb = lines[4];
        mel_M = lines[5];
        mel_m = lines[6];

        push!(results, eval(parse(major)));
        push!(results, eval(parse(minor)));
        push!(results, eval(parse(subprobs)));
        push!(results, eval(parse(rpkb)));
        push!(results, eval(parse(mel_M)));
        push!(results, eval(parse(mel_m)));
    end

    return results;
end
```

In [ ]:
```
function finalPreProc(filenames::Array{String})
    processAll(filenames);
end
```

In [ ]:
```
function finalMakeModel(filenames::Array{String}, modelname::String)

    model = readAllChords(filenames);

    writeModel(modelname, model);
end
```

In [ ]:
```
function finalGenerate(filename::String, mode::String, offset::Int, plength::Int, outname::String)

    model = readModel(filename);

    cpkb_M = model[1];
    cpkb_m = model[2];
    sub_probs = model[3];
    rpkb = model[4];
    mel_M = model[5];
    mel_m = model[6];

    prog = generateProgression(cpkb_M, cpkb_m, sub_probs, mode, plength);
    rhythms = generateRhythmProg(rpkb, plength);
    melody = generateMelody(mel_M, mel_m, mode, prog);

    outputProgression(prog, rhythms, melody, offset, outname);
end
```

Execution

Parameters

Populate these parameters to set up final execution functions below

In [ ]:
```
# List of input file names for preprocessing
filenames_pre = ["example1.mid", "example2.mid"];

# List of preprocessed input file names -- note that this list will have to be populated after running preprocess
ing on the input files and observing the output
filenames = ["Major_example1.mid", "Minor_example1.mid"];

# Name to give model text file, can be whatever, should have ".txt" extension
modelname = "model.txt";

# Mode for output ("Major" or "Minor")
mode = "Major";

# Interval by which to transpose output, any reasonable (resulting notes will be in a register humans can hear) n
egative or nonnegative integer
offset = 0;

# Number of measures to generate, works best as a multiple of 4
plength = 16;

# Name to give output MIDI file, can be whatever, don't include ".mid" extension
outname = "radsong";
```

Preprocessing Execution

In [ ]:
```
finalPreProc(filenames_pre);
```

Model Learning Execution

In [ ]:
```
finalMakeModel(filenames, modelname);
```

Generation Execution

In [ ]:
```
finalGenerate(modelname, mode, offset, plength, outname);
```

Appendix B

Corpus Description

Title	Composer	Key
Musette	Bach, J.S.	G Major
Little Prelude in F	Bach, J.S.	F Major
Ecossaise In G	Beethoven, L.	G Major
Fantasie Allegro	Blohm, S.	F Major
Pastorella	Blohm, S.	F Major
Prince of Denmark's March	Clarke, J.	D Major
Adagio	Mozart, A.	C Major
Canon in D	Pachelbel, J.	D Major
Suite in G Major: Prelude & Almand	Purcell, H.	G Major
La Joyeuse	Rameau, J.	D Major
#13 in A Minor	Bach, J.S.	A Minor
English Suite No. 06 in D Minor: Gavotte	Bach, J.S.	D Minor
Hungarian Dance No. 5	Brahms, J.	D Minor
Pavane	Faure, G.	F# Minor
Melpomene: Praeludium & Allemande	Fischer, J.	A Minor

Piano Solo	Tchaikovsky, P.	A Minor
Bourree	Telemann, G.P.	A Minor
Forever Rachel	Uematsu, N.	G Minor
Main Theme, Final Fantasy IV	Uematsu, N.	A Minor
Terra's Theme	Uematsu, N.	G Minor

Appendix C

Output Sheet Music

Perhaps some arcane technique exists, beyond the eldritch veil separating possibility from imagination, for coaxing simple paper to emit music using nothing more than printed ink. If so, it eludes the grasp of our most sophisticated science. In any case, the author has no access to such arts.

We are forced to encode music symbolically, instead. What follows are examples of Composobot's compositions, one in Major and one in Minor, encoded in sheet music. It isn't ideal, but all the information is there. These pages would sing, if wishing could but make it so.

Figure C.1: Composobot Output: Major

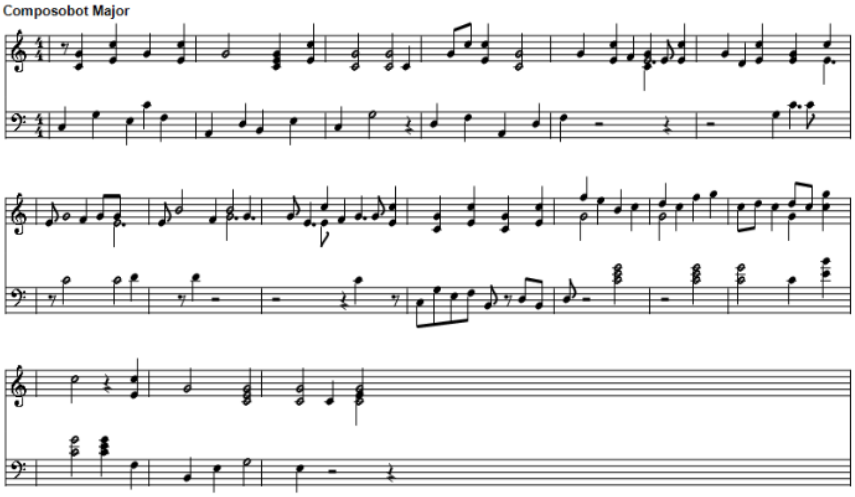


Figure C.2: Composobot Output: Minor